

IT Licentiate theses  
2004-004  
MRTC Report 2004/120

# On-Chip Monitoring for Non-Intrusive Hardware/Software Observability

MOHAMMED EL SHOBAKI

UPPSALA UNIVERSITY  
Department of Information Technology

**MRTC**  
MÄLARDALEN REAL-TIME  
RESEARCH CENTRE







UPPSALA  
UNIVERSITET

**On-Chip Monitoring for Non-  
Intrusive Hardware/Software  
Observability**

BY  
MOHAMMED EL SHOBAKI

September 2004

DIVISION OF COMPUTER SYSTEMS  
DEPARTMENT OF INFORMATION TECHNOLOGY  
UPPSALA UNIVERSITY  
UPPSALA  
SWEDEN

Dissertation for the degree of Licentiate of Technology in Computer Systems  
at Uppsala University 2004

# On-Chip Monitoring for Non- Intrusive Hardware/Software Observability

*Mohammed El Shobaki*

mohammed.el.shobaki@mdh.se

*Mälardalen Real-Time Research Center  
Department of Computer Science and Engineering  
Mälardalen University  
Box 883  
SE-721 23 Västerås  
Sweden*

<http://www.mrtc.mdh.se/>

<http://www.idt.mdh.se/>

© Mohammed El Shobaki 2004

ISSN 1404-5117

ISSN 1404-3041, ISRN MDH-MRTC-120/2004

Printed by Arkitektkopia, Västerås, Sweden

Distributed by the Department of Information Technology, Uppsala University, Sweden,  
and the Department of Computer Science and Engineering, Mälardalen University, Sweden

---

# Abstract

---

The increased complexity in today's state-of-the-art computer systems make them hard to analyse, test, and debug. Moreover, the advances in hardware technology give system designers enormous possibilities to explore hardware as a means to implement performance demanding functionality. We see examples of this trend in novel microprocessors, and Systems-on-Chip, that comprise reconfigurable logic allowing for hardware/software co-design. To succeed in developing computer systems based on these premises, it is paramount to have efficient design tools and methods.

An important aspect in the development process is observability, i.e., the ability to observe the system's behaviour at various levels of detail. These observations are required for many applications: when looking for design errors, during debugging, during performance assessments and fine-tuning of algorithms, for extraction of design data, and a lot more. In real-time systems, and computers that allow for concurrent process execution, the observability must be obtained without compromising the system's functional and timing behaviour.

In this thesis we propose a monitoring system that can be used for non-intrusive run-time observations of real-time and concurrent computer systems. The monitoring system, designated Multipurpose/Multiprocessor Application Monitor (MAMon), is based on a hardware probe unit (IPU) which is integrated with the observed system's hardware. The IPU collects process-level events from a hardware Real-Time Kernel (RTK), without perturbing the system, and transfers the events to an external computer for analysis, debugging, and visualisation. Moreover, the MAMon concept also features hybrid monitoring for collection of more fine-grained information, such as program instructions and data flows.

We describe MAMon's architecture, the implementation of two hardware prototypes, and validation of the prototypes in different case-studies. The main conclusion is that process level events can be traced non-intrusively by integrating the IPU with a hardware RTK. Furthermore, the IPU's small footprint makes it attractive for SoC designs, as it provides increased system observability at a low hardware cost.



*To my father and my mother*  
*Taisir and Rahmeh*  
*for their never ending love, support and sacrifices*





---

# Acknowledgments

---

This work would not have been possible without the support from many people whom I wish to thank. First, I would like to thank my dear supervisor Lennart Lindh, for giving me the opportunity to work for him and take me on as PhD Student. I've enjoyed working with you in many ways (not to forget our experiences together in preparing and selling fried herring during the summer festival in Västerås!). Second, I would like to sincerely thank Jan Gustafsson for taking the challenge to supervise me during the finishing of this work. You have been a source of inspiration, and I admire your dedication and patience with me. Third, I would like to thank Professor Hans Hansson, my main supervisor, for great reviewing and for monitoring my work actively, but non-interferingly!

I would also like to thank my colleagues and friends at the Computer Architecture Laboratory (CAL). Special thanks go to Joakim Adomat, for the creative discussions we've had throughout the years at the department, and also for designing many of the hardware prototypes I've used in my work. I'm also grateful to Johan Stärner for helping me with various issues regarding hardware design. Thanks Filip Sebek for your kind feedback on my work, and for making use of my research results in your own experiments – it has been valuable to me.

I would also like to acknowledge the efforts of the following people for their valuable support with implementing various parts of the software used in my work: Jeroen Heijmans, Andreas Malmquist, Adil Al-Wandi, Mehrdad Hessadi, Mladen Nikitovic, Johan Andersson and Toni Riutta.

Many thanks goes also to my fellow director of undergraduate studies, Åsa Lundkvist, and Monica Wasell, for backing me up during the final work with this thesis. Their support has been invaluable to me.

At the department where I work there are many people whom I wish to thank. It would be a bit impractical to list them all, even though I care for them all. There are however some people whom I wish to thank especially: Harriet Ekwall for creating such a homely working environment, Henrik Thane for constructive comments and ideas he had in relation to my initial work, and Thomas Larsson for the sincere moral support he has given me.

I'm also indebted to my parents, my brothers and sisters, for the wonderful moral support they have given me. Sincere thanks goes also to my family-in-law, the Masud family, for their wonderful support in many ways, including the nice meals and pastries they have stuffed me with during my stressful moments.

Last but not least, I'd like to give a huge virtual bunch of roses to my love in life, my wife Mouna, for her sweetness, devotion and support throughout my work. I love you!

This work has been financially supported by the KK Foundation, Vinnova (formerly Nutek), and Mälardalen University, for which I am very grateful.

Mohammed El Shobaki  
A beautiful day in September, 2004

---

# Contents

---

<b>I</b>	<b>Thesis</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	4
1.2	Thesis Outline . . . . .	5
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Embedded and Real-Time Systems . . . . .	7
2.1.1	Concurrency, Tasks, and Processes . . . . .	9
2.1.2	Real-Time Operating Systems . . . . .	9
2.1.3	Real-Time Kernel . . . . .	10
2.2	Multiprocessor and Distributed Systems . . . . .	10
2.3	Testing, Debugging and Performance Analysis . . . . .	12
2.3.1	Debugging and Testing . . . . .	12
2.3.2	Performance Analysis . . . . .	12
2.4	Monitoring . . . . .	13
2.4.1	Monitoring Abstraction Levels . . . . .	14
2.4.2	Types of Monitoring Systems . . . . .	16
2.4.3	The Probe Effect . . . . .	20
<b>3</b>	<b>Problem Formulation</b>	<b>23</b>
<b>4</b>	<b>Contributions</b>	<b>25</b>
<b>5</b>	<b>Summary of Papers</b>	<b>29</b>
5.1	Summary of Paper A . . . . .	29
5.2	Summary of Paper B . . . . .	30
5.3	Summary of Paper C . . . . .	31
<b>6</b>	<b>Related Work</b>	<b>33</b>
6.1	Monitoring Real-Time Kernels . . . . .	33
6.2	Hardware Monitoring Systems . . . . .	35
6.3	Hybrid Monitoring Systems . . . . .	39

6.4	On-Chip Techniques . . . . .	40
<b>7</b>	<b>Conclusions</b>	<b>43</b>
<b>8</b>	<b>Future Work</b>	<b>45</b>
	<b>Bibliography</b>	<b>47</b>
<b>II</b>	<b>Included Papers</b>	<b>53</b>
<b>9</b>	<b>Paper A:</b>	
	<b>A Hardware and Software Monitor for High-Level System-on-Chip Verification</b>	<b>55</b>
9.1	Introduction . . . . .	57
9.2	MAMon . . . . .	59
9.2.1	The Probe Unit . . . . .	60
9.2.2	Host interface . . . . .	62
9.2.3	The tool environment . . . . .	63
9.3	An Ideal Example: Monitoring a Hardware Real-Time Kernel	64
9.4	Current and Further Work . . . . .	65
9.5	Conclusions . . . . .	66
	Bibliography . . . . .	66
<b>10</b>	<b>Paper B:</b>	
	<b>On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems</b>	<b>69</b>
10.1	Introduction . . . . .	71
10.2	A Real-Time Multiprocessor Architecture - SARA . . . . .	73
10.2.1	RTU - Real-Time Kernel in Hardware . . . . .	74
10.2.2	A SARA CompactPCI System . . . . .	75
10.3	A Monitoring System for Hardware-Accelerated RTOSs . . . . .	76
10.3.1	Overview . . . . .	76
10.3.2	The Integrated Probe Unit . . . . .	77
10.3.3	Events . . . . .	80
10.3.4	Performance and FIFO Dimensioning . . . . .	81
10.3.5	The Monitoring Application Framework . . . . .	82
10.4	Physical Hardware Implementation . . . . .	84
10.4.1	The Hardware Prototype . . . . .	85
10.4.2	Physical Footprint . . . . .	85

10.5 Prototype Evaluation . . . . .	86
10.6 Conclusions . . . . .	87
Bibliography . . . . .	89
<b>11 Paper C:</b>	
<b>MAMon - A Multipurpose Application Monitor</b>	<b>93</b>
11.1 Introduction . . . . .	95
11.1.1 Related documents . . . . .	95
11.2 Overview of MAMon . . . . .	95
11.3 The Integrated Probe Unit . . . . .	97
11.3.1 Entity interface . . . . .	97
11.3.2 The Host Interface . . . . .	99
11.4 The MAMon Application Framework . . . . .	103
11.4.1 Connection with hardware . . . . .	103
11.4.2 The main program . . . . .	103
11.5 Framework Software Architecture . . . . .	106
11.5.1 Packages . . . . .	106
11.5.2 Meeting requirements . . . . .	106
11.5.3 Architecture Overview . . . . .	109
11.5.4 The retrieval mechanism . . . . .	112
11.6 Framework Programmer's Guide . . . . .	117
11.6.1 General . . . . .	117
11.6.2 Adding a new plug-in . . . . .	120
11.6.3 Using the event definitions file . . . . .	125
11.6.4 Changing the DBMS . . . . .	125
11.6.5 Specific plug-ins . . . . .	126
11.7 MAMon Tool Desktop User Guide . . . . .	128
11.7.1 Quick start . . . . .	128
11.7.2 Reference guide . . . . .	129
11.7.3 Plug-in Tools . . . . .	140
Bibliography . . . . .	146
<b>A Patent</b>	<b>149</b>
A.1 Field of the invention . . . . .	153
A.2 Prior art . . . . .	153
A.3 Summary of the invention . . . . .	154
A.4 Description of the drawings . . . . .	156
A.5 Description of embodiments . . . . .	157
A.6 Claims . . . . .	161

A.7 Drawings . . . . . 162

---

# List of Publications

---

This Licentiate<sup>1</sup> thesis is a summary of the following three papers. References to the papers will be made using the letters associated with the papers.

- A. Mohammed El Shobaki and Lennart Lindh, A Hardware and Software Monitor for High-Level System-on-Chip Verification, In proceedings of the IEEE International Symposium on Quality Electronic Design, San Jose, CA, USA, March 2001.
- B. Mohammed El Shobaki, On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems, In proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA), Tokyo, Japan, March 2002.
- C. Mohammed El Shobaki and Jeroen Heijmans, MAMon - A Multipurpose Application Monitor, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-121/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden, September 2004.

Besides the above papers I have authored and co-authored the following scientific publications:

- I. Mohammed El Shobaki, Observability in Multiprocessor Real-Time Systems with Hardware/Software Co-Simulation, In Swedish National Real-Time Conference SNART'99, Linköping, Sweden, August 1999.
- II. Mohammed El Shobaki, Verification of Embedded Real-Time Systems using Hardware/Software Co-simulation, In proceedings of the 24th Euro-micro Conference, Vol. I, pp. 46-50, Västerås, Sweden, August 1998.
- III. Lennart Lindh, Johan Stärner, Johan Furunäs, Joakim Adomat, and Mohammed El Shobaki, Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems, In Seventh Swedish Workshop on Computer Systems Architecture, Chalmers, Göteborg, Sweden, June 1998.

---

<sup>1</sup>A Licentiate degree is a swedish degree halfway between MSc and PhD.





# **I**

## **Thesis**



---

# Chapter 1

## Introduction

---

As human beings we strive for comfort and easy living. Therefore we build advanced devices and machines that can automate hard duties. As humans we need to communicate with each other and we need to be entertained. Therefore we build telecommunication systems, television, home cinema, and computer games. To discover far places and meet other people we need to travel. Therefore we build automobiles, ships and air craft. In almost each of these inventions by the modern human we can find computer systems, that is, intelligent pieces of electronics that do what we program them to do. Thus, our everyday lives are becoming increasingly more dependant on these systems, and we take for granted that they work properly and safely.

As a concrete example, take for instance the features of a modern car. Figure 1.1 illustrates features that typically utilize a computer, for example, the engine control computer that optimize performance and fuel combustion, the computer that detects collisions and activate the airbags in the event of a crash, the computer that regulates the interior climate based on passenger preferences and exterior climate conditions, the computer that controls the braking system for maximum efficiency, and more. In fact, a modern car may contain up to a 100 computer systems that together orchestrate all these features.

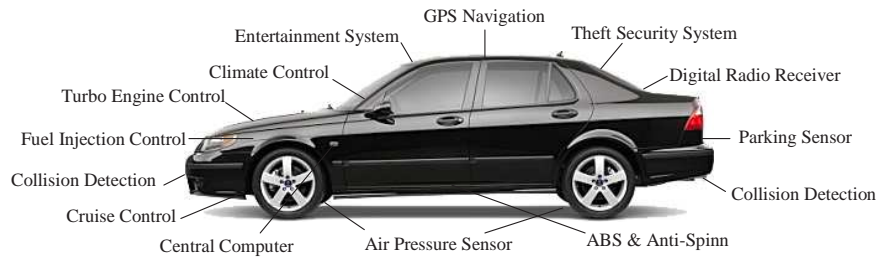


Figure 1.1: Example of features of a modern car

This thesis is about methods for observing the behaviour of a computer system. These observations are necessary for a number of reasons, including the need for testing and optimisation during the development of computer systems. The thesis proposes a concept for carrying out certain types of observations without disturbing the natural behaviour of the observed computer, for reasons which will be discussed in the following chapters.

In Section 1.1 we will present the overall motivations for this work, and in Section 1.2 we outline the thesis contents.

### 1.1 Motivation

Today's computer-based products are complex and require extensive efforts to design and test. They are complex because they comprise many components, complex software and hardware, and features a lot of functionality. This is a trend which is clearly seen in the consumer electronics market, and in state-of-the-art industrial systems. The development of these products tends to be as challenging as it is increasingly time-consuming, expensive, and error-prone. Therefore, the developers need to cut down the development time and improve quality, which in turn, demands better tools and development methodologies.

One important aspect in the development process is *observability*, i.e., the ability to observe the system's behaviour at various abstraction levels in the design. These observations are required for many reasons, for instance, when looking for design errors, during debugging, during optimisation of algorithms, for extraction of design data, and a lot more. Observability is however not an issue restricted to development purposes only, it may also be necessary after the deployment of products as well, e.g., for error recovery, for surveillance

issues, for collection of statistical measurements (e.g. concerning the use of a product), etc.

We characterize the quality of observability as: *good* (or high) if the system allows for detailed and accurate analysis of all of its components, and *poor* (or low) if the system is obstructive and hard to analyse confidently.

In essence, this thesis is motivated by the needs from industry in seeking better observability for complex computer systems based on state-of-the-art hardware and software architectures.

## 1.2 Thesis Outline

The thesis is divided into two parts, where the first part (part I) gives an introduction to the research area, describes the research problems, presents the thesis contributions, conclusions, etc. More specifically: Chapter 2 lays the background which the subsequent discussions will proceed from. Chapter 3 presents the problems we have focused on. In Chapter 4 we present the main contributions of this work. Chapter 5 summarises the papers included in the second part of the thesis. In Chapter 6 we present relevant related work. Chapter 7 presents the thesis conclusions, and in Chapter 8 we give some directions on future work.

In the second part (part II) of the thesis we have appended the included papers, Paper A - C.

Finally, in the appendix we have enclosed a Swedish-registered patent that constitutes one of our contributions (described in Chapter 4).



---

## Chapter 2

# Background

---

This chapter presents the basic concepts used in the thesis. The concepts and their related terms will be assumed to be familiar to the reader in the discussions throughout the thesis.

### 2.1 *Embedded and Real-Time Systems*

An *embedded system* is typically a product which includes a computing system. The product is said to "embed" the computing system inside. Embedded systems do not necessarily look like computers, however it is typical that embedded systems interact with their environment. For instance, a mobile phone is regarded an embedded system: it reacts on incoming calls, user input, cell roaming, etc. A talking doll is another example, since the doll might express a message based on which part of its body, or button, has been pushed, or if it gives a response to a playing child's voice.

A *real-time system* is a system that interacts with its environment in a timely constrained manner. The real-time system must produce results within specified time limits. A computation result (or actuation) must be delivered neither too late, nor too early. The criticality of violated timing constraints, or *missed execution deadlines*, classifies real-time systems into *hard* or *soft* real-time systems [But97]. Timing failures in a hard real-time system are considered hazardous and very critical and should never be allowed. Examples on hard real-time requirements can be found in automotive and avionic systems, medical equipment, military systems, energy and nuclear plant control systems. On the contrary, the requirements in soft real-time systems are not so critical and may

tolerate timing constraint violations, either by discarding the produced results or by allowing a degraded quality. Soft real-time requirements can be found in telecommunication systems, audio and video applications, streaming media, airline reservation systems, etc.

A typical real-time system (see Figure 2.1) consists of a controlling subsystem (the computer), and the controlled subsystem (the physical environment). The interactions between the two subsystems can be described by three main operations:

- Sampling
- Processing
- Responding

The computer subsystem continuously samples data from the physical environment. Sampled data is then immediately processed by the computer subsystem, and a proper response is sent to the physical environment. All three operations must be performed within the required timing constraints. For example, it is imperative that an air bag control system in an automobile responds within set timing constraints in the event of a crash. The response must neither be too late (being non-effective), or too early (risking hazardous manoeuvring of the car).

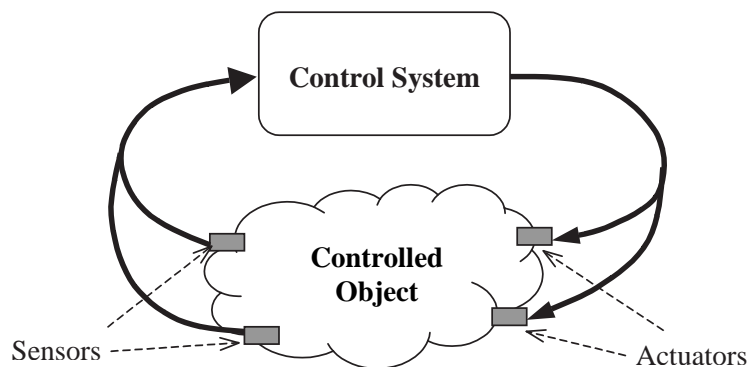


Figure 2.1: A real-time system



### 2.1.1 Concurrency, Tasks, and Processes

The real-time computing software is in its simplest form implemented as one big program loop. Typically, such programs can be found in *Programmable Logic Controllers* (PLCs) which are used to control relatively simple (industrial) applications [NSMT<sup>+</sup>00]. PLC programs are often realised as loops that include instructions to read input data (e.g. from sensors), perform logical processing on the input, and write out data (e.g. to actuators or relays).

When the controlled environment is more complex, the real-time software may need to be divided into several *tasks*. A task, also called *thread*, is an independent sequence of program instructions which may execute concurrently with other tasks (multitasking) on the same real-time computer. Tasks execute under the control of a *Real-Time Operating System* (see below) which also manages the computer resources (processor and memory), inter-task communication, synchronisation, and I/O. Tasks normally share the memory space – both instructions and data – with other tasks. The shared memory is typically also used for communication and synchronisation with other tasks.

A software *process* is a special case of a task which have an own protected memory space, i.e. it does not share memory with other processes. The process may be seen as a standalone program acting as though it owns the computer for itself. Moreover, a process may internally be represented by one or more concurrent tasks that share execution within the process. Memory-protection between processes is usually implemented using a hardware *memory management unit* (MMU) which checks accesses to privileged memory. Whenever processes needs to communicate (*Inter-Process Communication*), messages must be passed via the RTOS (using *message-passing*) which normally hides copying of message data between processes' memory space.

From now on the terms *task* and *process* will be used interchangeably in the text, unless they are explicitly distinguished.

### 2.1.2 Real-Time Operating Systems

A *Real-Time Operating System* (RTOS) is an operating system specially intended for real-time systems, that allows easier design, execution, and maintenance of real-time systems and applications. The use of an RTOS simplifies the design process by providing the developer with a uniform programming interface to the underlying computer hardware. In this way, the developer may focus on designing the application rather than bothering about the details and structure of the computer hardware. The main responsibility of an RTOS during run-time is to manage the available computing resources so that application

tasks may share, and synchronise their use of, these resources in a way that timeliness is ensured. Timeliness is ensured by *scheduling* (see Section 2.1.3), which is the main technique used to guarantee availability of resources at the right time to the tasks.

Another responsibility of an RTOS is the management of communication between processes, and synchronisation of resource utilisation. A process that wants to communicate a message to another process usually does this by invoking a system-call to the RTOS which then takes care of data copying and notification of message arrival to the addressed process(es). In the case of resource synchronisation, the RTOS typically administrates certain data structures (*mailboxes*, *queues*, *mutexes*, *semaphores*, etc. [Lab02]) that organises process admission to the shared resources.

### 2.1.3 Real-Time Kernel

The core of an RTOS is the *Real-Time Kernel* (RTK). This component manages the scheduling of process execution on the available CPU-resources in the system. In single-processor systems the processes will *time-share* the same processor, and in a multiprocessor system the processes will be distributed over the processors. The time-shared execution follows a *scheduling scheme* which is tailored to fit the design requirements. For instance, one scheduling scheme is that every process should get equal time shares for execution. Another scheme may be that processes execute based on their *priority*, i.e., the process with highest priority is allowed to execute before processes with lower priorities. There are also schemes that follow execution schedules that are defined pre-run-time, so-called *static schedules*, which contains activation times (*absolute* or *relative time*) for each process [Liu00]. The RTK may also be responsible for the scheduling of other resources than the CPU [Lab02].

A typical RTK is implemented in software as part of the RTOS. There exist however implementations of the RTK in hardware [Lin92, AFLS96, MRS<sup>+</sup>90, NUI<sup>+</sup>95] (described also in the included papers of Part II). The main benefit with hardware-implemented RTKs is that they execute in parallel with the CPU(s) in the system, i.e. like a *co-processor*, which results in a performance acceleration of the RTOS' operation in some type of systems [Fur00].

## 2.2 Multiprocessor and Distributed Systems

Over the past years we have seen a trend towards parallel computing as opposed to single processor systems. There are several reasons for this trend, including

the following:

- Physical speed limit – Processor manufacturing is facing physical limits such as line-widths on silicon, limit in speed of light (high frequencies), signal quality, etc.
- Special purpose processing – In some systems it is better to partition and distribute a computation over a set of special purpose processors, rather than using one general-purpose processor. For example, a 3D graphics computation is best done using an array processor and a DSP rather than using a general CPU.
- Fault tolerance and availability – By increasing the number of processing elements, computer systems can be made more fault-tolerant in the event of failures.
- The Internet – There is no doubt that the Internet has greatly contributed to the demand for higher performance and throughput in communication applications. Large database systems are today a big market for multi-processing.

Designing multiprocessor systems and applications is however not trivial, and requires deep understanding of parallelism and problems related to concurrency. When a program is partitioned into portions that are allowed to execute in parallel, i.e. processes, it is usually necessary to communicate data between them in order to fulfil a computation. This inter-process communication usually requires synchronisation of the involved parts. For instance, a *process A* that intends to read a message from *process B* must be synchronised with the availability of the message from *process B*. The typical errors in multiprocessor systems are particularly related to communication and synchronisation [Gai86].

There are various meanings for what is to be considered a multiprocessor system. Some texts depend their definition on the communication media between the processors, e.g. shared-memory, distributed memory, or communication over a network. Other texts rely the definition on the usage of a common (global) clock. In the latter meaning, a system with different clocks (one for each processor) is considered a distributed system. In this text the definition is; *if two or more processing elements are used in a computation of a program, it is considered a multiprocessor system*. Hence, we do not distinguish between the terms multiprocessor and distributed systems.

## 2.3 Testing, Debugging and Performance Analysis

### 2.3.1 Debugging and Testing

Debugging, as defined in the ANSI/IEEE standard glossary of software engineering terms [MH89], is "*the process of locating, analysing, and correcting suspected faults*". A fault is defined to be the direct cause of some error. Since the occurrence of errors can have different reasons, they are usually not predictable, and therefore we must locate them using *debuggers*. A debugger is a tool which helps the designer to examine suspected errors in a program, and eventually also remedy the errors. The term *cyclical debugging* is commonly used to describe debugging as an iterative process, in which the debugger is used to find and correct errors, over and over again, until no more errors can be found.

Testing and debugging are similar activities with respect to finding errors. However, testing is more of an automated process of exposing different input to the system under test, and evaluating its results (output). The objective is to find input data, or patterns of data, that cause erroneous results [SVS<sup>+</sup>88]. The faults that are found during the testing process are then put under observation in a debugger.

In real-time systems, errors may also occur in the time-domain. Real-time systems are therefore harder to debug than non-real-time systems. The ability to track down timing-related errors was largely an unexplored area until the early 1980's. Glass (in [Gla80]) reported a significant lack of effective tools in the emergence of real-time systems development and referred to the problem as the "*lost world*" of software testing and debugging. Today, various debugging systems and methods have been developed in order to address timing-related issues [TFC90a, JRR94, TSHP03].

### 2.3.2 Performance Analysis

While removal of errors is an important part of the design process, others are implementation optimisation and fine-tuning of algorithms. To pursue such activities the designer needs to analyse behaviour and performance of the developed system, its components and sub-components. Performance analysis is of importance to find performance bottlenecks, and to extract design-parameters such as execution times, response times, communication delays of various kind, and so on. The extraction of design-parameters is for instance valuable to task scheduling analysis and estimates for resource allocation.

In the next section we will describe how testing, debugging and performance

analysis may be facilitated through the use of *monitoring*.

## 2.4 Monitoring

Monitoring is *the process of gathering information about a system* [TBHS96, MH89]. We gather information which normally cannot be obtained by studying the program code only. The collected information may be used for program testing, debugging, task scheduling analysis, resource dimensioning, performance analysis, fine-tuning and optimisation of algorithms. The applicability of monitoring is wide, and so is the spectrum of available monitoring techniques. In this section we give a general presentation of a monitor, and describe different monitoring systems, the type of information collected by monitors, and the problem-related issues with monitoring.

In essence, a monitor works in two steps: *detection* (or triggering) and *recording*. The first operation refers to the process of detecting the object of interest. This is usually performed by a *trigger object* that is inserted in the system, which when executed, or gets activated, indicates an event of interest for recording. The latter operation, recording, is the process of collecting events and saving them in buffer memory, or communicate them to external computer systems for the purpose of further analysis or debugging. An event is a record of information which usually constitutes the object of interest together with some additional meta data regarding that object (e.g. the time when the object was recorded, the object's source address, task/process ID, CPU node, etc.). The type of monitored objects depend on the level of abstraction which the user is interested in. Section 2.4.1 below describes different abstraction levels that are associated with program execution. The trigger object may be an instruction, or a function, that is inserted in the software. It may also be a physical sensor, or *probe*, connected with physical wires in the hardware, such as CPU address, data, and control buses.

An important issue regarding the monitoring process is the amount of *execution interference* that may be introduced in the observed system due to the involved operations of a monitor. This execution interference, or *perturbation*, is unwanted because it may alter the true behaviour of the observed system, in particular such systems that are inherently timing-sensitive such as real-time and distributed systems. We will return to this issue in the discussion on *probe-effects* in Section 2.4.3.

### 2.4.1 Monitoring Abstraction Levels

Software execution may be monitored at different levels of abstraction as the information of interest is different in levels of detail. Higher-level information refers to events such as inter-process communication and synchronisation. In contrast, lower-level information refers to events such as the step-by-step execution trace of a process. The execution data collected at the process level includes the process state transitions, communication and synchronisation interactions among the software processes, and the interaction between the software processes and the external process. The execution data collected at the function level includes the interactions among the functions or procedures within a process. The user can isolate faults within functions using the function-level execution data. In this section, the different levels of abstraction in software execution are identified.

#### 2.4.1.1 System Level

The system-level may be seen as the user's, or the real-world, view of the computer system. It abstracts away all implementation details and only provides information that is relevant to the system's user (or to the real-world process). For instance, the *press* of a button in a car's instrument board, and the *activation/deactivation* of the car's *Traction Control System* (anti-spinning system) feature, would be considered as system-level events (see Figure 2.2). This level of information is normally useful for system-test engineers during the final steps in the development process.

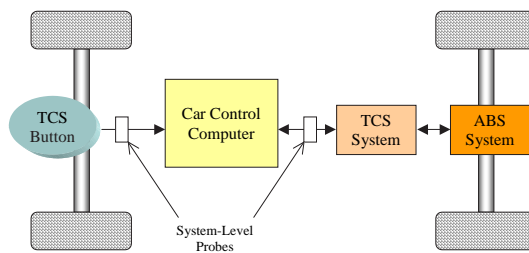


Figure 2.2: Monitoring at the system level of abstraction

#### **2.4.1.2 Process and OS Level**

To monitor program execution at the process level, we consider a process as a black box which can be in one of the three states: *running*, *ready*, or *waiting*. A process changes its state depending on its current state and the current events in the system. These events include interactions among the processes and the interactions between the software processes and the real world. The events that directly affect the program execution at the process level are distinguished from those events that affect the execution at lower levels. Assigning a value to a variable, arithmetic operations, and procedure calls, for instance, are events that will not cause immediate state changes of the running process. Inter-process communication and synchronisation are events that may change a process' running status and affect its execution behaviour. The following events are typically considered as process level events:

- Process Creation
- Process Termination
- Process State Changes
- Process Synchronisation
- Inter-process Communication
- External Interrupts
- I/O Operations

#### **2.4.1.3 Functional Level**

The goal of monitoring program execution at the function level is to localise faulty functions (or procedures) within a process. At this level of abstraction, functions are the basic units of the program model. Each function is viewed as a black box that interacts with others by calling them or being called by them with a set of parameters as arguments. So the events of interest are function calls and returns. The key values for these events are the parameters passed between functions.

#### **2.4.1.4 Instruction Level**

The instructional level of abstraction refers to the step-by-step execution of CPU-instructions. It is, from a software perspective, regarded as the lowest

level of abstraction of a program for a modern CPU<sup>1</sup>. To monitor each executed instruction is, however, a heavy duty on any monitor since it requires at least the CPU-performance of the system being observed, and the collected amounts of event traces are too huge to be of practical use. Instead, it is sufficient enough to monitor just those instructions that affect the execution path of a program, e.g. conditional branches, traps, exceptions, etc. Using this information in combination with the software's source, or object code, it is possible to reconstruct the execution behaviour. For many programs<sup>2</sup>, such a method reduces the amount of recorded data with several orders of magnitude.

## 2.4.2 Types of Monitoring Systems

Monitoring systems for software or system-level analysis are typically classified into three types: 1) *software monitoring systems*, 2) *hardware monitoring systems*, and 3) *hybrid monitoring systems*. In the following we will describe each type of system. Chapter 6 gives a more detailed presentation of monitoring systems that relates to our work on hardware and hybrid monitoring.

### 2.4.2.1 Software Monitoring Systems

In this category of monitoring systems, only software is used to instrument, record, and collect information about software execution. Software monitoring systems offer the cheapest and most flexible solution where a common technique is to insert instrumentation code at interesting points in the target software. When the instrumentation code is executed the monitoring process is triggered and information of interest is captured into trace buffers in target system memory. The drawbacks of instrumentation is the utilisation of target resources such as memory space and processor execution time.

Below is a more detailed description of a specific monitoring tool, called *StethoScope*, which serves as an example on how a typical software monitor operates.

#### **StethoScope**

StethoScope [Baw99] by *Real-Time Innovations Inc.* is a commercially avail-

---

<sup>1</sup>In earlier days an instruction was seen as a composition of sub-instructions, called microcode, which together carried out the different operations that occur inside the CPU-core (e.g. memory load/store operation, register shifts, ALU-operations and bit-tests, etc). Today however, micro-coding is rarely done by software designers, though there exists application specific CPUs that allows micro-coding.

<sup>2</sup>It is widely known that many programs spend (very roughly) 90% of their time in about 10% of their code; 10% of static instructions account for 90% of dynamic instructions.



able tool for monitoring real-time systems. The monitoring process is claimed to be non-intrusive since the sampling of the system is limited to only reading variables from the application memory. Their definition of non-intrusive monitoring means, however, that the application software does not require modification.

StethoScope comprise a set of monitoring tasks on the target, and a GUI on a host computer, see Figure 2.3. The monitoring tasks are compiled and linked together with the application. During program execution, the *Sampler Task* periodically awakes and copies the currently monitored variables (denoted *signals* in the GUI) from their addresses in the application to the *Target Buffer*. Later the *ScopeLink daemon* copies the Target buffer to the GUI's *Live Buffer*. The user can at any time, via the StethoScope GUI, choose the signals (variables) that will be monitored, and change data collection parameters, for example the rate at which data is collected. Such requests are handled by the *ScopeProbe daemon* which in turn updates internal data structures that control the monitoring process.

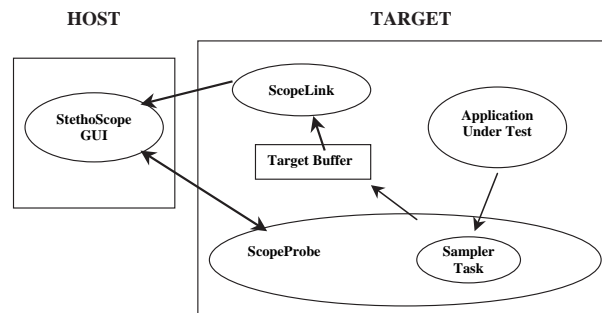


Figure 2.3: The components of StethoScope's architecture (ref. [StethoScope1999])

The execution of the application is of course disturbed during the periodical copying of memory. *ScopeProbe's* *Sampler* task runs at the highest priority and needs to interrupt the application to perform its copying function. Thus, StethoScope's monitoring process cannot be claimed to be non-intrusive in the sense we have discussed in the previous section. However, StethoScope calls this *non-intrusive asynchronous* monitoring. It is asynchronous in the sense that samples are taken at specific time intervals, i.e. they are not co-ordinated

with the events in the program. For example, variables can be assigned values several times (e.g. in a loop) between each invocation of the Sampler task. This way of monitoring is also said to be *discontinuous*. Another disadvantage with StethoScope's asynchronous monitoring is that it can only sample static or global variables. Stack variables may be out of scope when the sampling occurs.

In order to monitor stack variables, the StethoScope system offers a synchronous monitoring model which, however, requires instrumented code. The instrumented code has calls to StethoScope's ScopeProbe API inserted at the locations where synchronous sampling is required. A call to the API function *ScopeCollectSignals()* will force sampling to occur in the same scope (immediately). Thus, stack variables can be monitored. The advantages with synchronous monitoring are precise control of sampling relative to program events and consistent data, since the variables are always sampled at the same point in the program.

#### 2.4.2.2 Hardware Monitoring Systems

In this category of monitoring systems, only hardware (custom or general) is used to perform detection, recording and collection of information regarding the software. For this to work, the target system must lend itself for observations by external means (the monitoring hardware).

The primary objective of hardware monitoring is to avoid, or at least minimize, interference with the execution of the target system. A hardware monitoring system is typically separated from the target system, and thus, does not use any of the target system's resources. Execution of the target software is monitored using passive hardware (or probes) connected to the system buses and signals. In this manner, no instrumentation of the program code is necessary. Hardware monitoring is especially useful for monitoring real-time and distributed systems since changes in the program execution time are avoided.

In general, the operation of monitoring hardware can be described by the three steps (see Figure 2.4): *event detection*, *event matching*, and *event collection*. In the first step, detection, the hardware monitor listens continuously on the signals. In the second step, the signal samples are compared with a predefined pattern which defines what to be considered as events. When a sample matches an event-pattern, the process triggers the final step, collection, where the sampled data is collected and saved. The saved samples may be stored locally in the monitoring hardware, or be transferred to a host computer system where usually more storage capacity can be obtained.

Apart from the advantage of avoiding target interference, are the typical

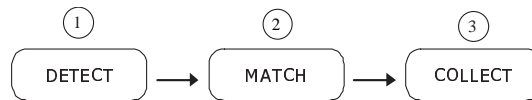


Figure 2.4: Hardware monitoring steps

precision and accuracy of hardware monitors. Since the sole duty of a hardware monitor is to perform monitoring activities (usually at equal or higher system speed than the target's) the risks of losing samples are minimized.

A disadvantage of hardware monitors is their dependency on the target's architecture. The hardware interfaces, and the interpretation of the monitored data must be tailored for each target architecture it is to be used in. Thus, monitoring solutions using hardware are more expensive than software alternatives. Moreover, a hardware monitor may not be available for a particular target, or takes time to customize, which may increase the costs further in terms of delayed development time.

Another problem with hardware is the integration and miniaturisation of components and signals in today's chips which renders difficulties in reaching information of interest, e.g. cache-memory, internal registers and buses, and other on-chip logic. To route all internal signals out from a chip may be impossible because of limited pin counts.

In general, hardware monitoring is used to monitor either hardware devices or software modules. Monitoring hardware devices can be useful in performance analysis and finding bottlenecks in e.g. caches (accesses/misses), memory latency, CPU execution time, I/O requests and responses, interrupt latency, etc. Software is generally monitored for debugging purposes or to examine bottlenecks, load-balancing (degree of parallelism in concurrent and multiprocessor systems), and deadlocks.

#### 2.4.2.3 Hybrid Monitoring Systems

Hybrid monitoring uses a combination of software and hardware monitoring and is typically used to reduce the impact of software instrumentation alone. A hardware monitor device is usually attached to the system in some way, e.g. to a processor's address/data bus, or on a network, and is made accessible for instrumentation code that is inserted in the software. The instrumentation is typically realised as code that extracts the information of interest, e.g. variable data, function parameters, etc., which is then sent to the monitor hardware.

For instance, if the monitor hardware has memory-mapped registers in the system, the instrumentation would perform data store operations on the monitor's memory-addresses. The hardware then proceeds with event processing, filtering, time-stamping, etc., and then communicates the collected events to an external computer system. This latter part typically resembles the operation of a pure hardware monitor. The insertion of instrumentation code also resembles the technique used in a software monitoring system; i.e. it can either be done manually by the programmer, automated by a monitoring control application or by compiler directives.

### 2.4.3 The Probe Effect

Instrumentation of programs, also called "probing", is convenient because it is a general method which technically is applicable in many systems. For concurrent programs however, the delay that is introduced by the insertion of additional instructions may alter the behaviour of the program. The *probe-effect*, which originates from *Heisenberg's Uncertainty Principle*<sup>3</sup> applied to programs [Gai86, MH89], may result in that either a non-functioning concurrent program works with inserted delays, or a functioning program stops working when the inserted delays are removed. This can also be seen as a difference between the behaviour of a system being tested and the same system not being tested. Typical errors related to the probe-effect are synchronisation errors in regions containing critical races for resources [Gai86].

Not only may concurrent programs suffer from the probe-effect, but also real-time systems are concerned since they are inherently sensitive to timing disturbances, especially if deadlines are set too tight (i.e. non or low-relaxed worst-case execution times). Consequently, distributed/parallel real-time systems are most sensitive to probe-effects. This is one important reason why testing and debugging (using monitoring) of real-time systems (particularly distributed real-time systems) is so difficult [Tha99, TBHS96, MH89]. Hence, probe-effects must be avoided in the development of real-time systems. There are basically three approaches to eliminate the probe-effect:

- *Leave the probes in the final system.* In this approach the probes that have been used during development are left in the final product. This way we

---

<sup>3</sup>Bugs that relate from probe-effects are in some texts referred to as "Heisenbugs" after the Heisenberg Uncertainty Principle from physics. This principle states that the instrumentation used to measure something, no matter how non-intrusive one may think it is, will always perturb the object being measured and result in an inaccurate measurement.

avoid behavioural changes due to removal of probes. The disadvantage is of course that the final system may suffer from inferior performance.

- *Include probe-delays in schedulability analysis.* In real-time systems design it is straightforward to include the probes in the execution time of the program, i.e. dedicate resources (execution time, memory, etc) to probes. However, this method does not guarantee the ordering of events, it only provides enough execution time to compensate for the inserted delays.
- *Use non-intrusive hardware.* Bus-snoopers and logic analysers are typical examples of passive hardware which do not interfere with the system. Other techniques are the use of multi-port memories, reflective memory, and use of special hardware. There are also hybrid monitoring systems which utilise hardware support together with software instrumentation. The disadvantage of this solution may be higher development and product costs due to extra hardware.



---

## Chapter 3

# Problem Formulation

---

In the previous chapter we have discussed the necessity of observability of the components of computer systems, during development and after deployment. We will now describe our main research problems in terms of three central questions that the thesis will provide answers to.

In our research group we are interested in exploiting the use of hardware parallelism to improve performance, as well as the determinism, of RTOS functions in real-time computers. As a result of this research we have developed several hardware implementations of an RTK [AFLS96, LSF<sup>+</sup>98, LKF99], with various features that range from simple priority-based task-scheduling for single-processor systems, to support scheduling, IPC, and interrupt management, for multiprocessor systems. In realising these systems we encountered difficulties in tracking down bugs that appeared at run-time, mainly because it was nearly impossible to determine if the bugs were located in the hardware RTK, or in the software that made use of it. Moreover, for the same reasons as when tracking bugs, it was not straightforward to analyse the system's performance and the possible execution speed-ups with hardware-acceleration. These struggles led us to the formulation of the first question:

**Question 1 (Q1).** *How can we observe, analyse, and visualize the run-time behaviour of processes in single- and multiprocessor computer systems that employ a hardware RTK?*

Note that in Q1 we focus only on the run-time behaviour, i.e. we are not interested in observing a simulated model of the system. We also discard so-

lutions that imply restricted, or lowered, execution speeds. This latter requirement rules out emulation systems and logic analysers, since they typically do not allow for full execution speeds [Ref till ngn känd survey :)].

The following question is related to Q1 in that a solution to the observation problem should not result in an altered behaviour of the system, or a change of the system's timing characteristics. The answer to this question is in fact the same as providing a solution to Q1 without introducing probe-effects (see Chapter 2). The question is justified because there exists attempts to utilise software tasks (special monitor/debug tasks) that polls the hardware contents of the RTK via its register interface. Hence, we formulate the second question as follows:

**Question 2 (Q2).** *Can we develop a solution to Q1 without perturbing the functional behaviour and timing properties of the observed system?*

The answers to Q1 and Q2 respectively are provided in Contribution A and B, and partly through Contribution E (see Chapter 4).

While Q1 and Q2 only concerns observations at the process-level, i.e. such information that would only require monitoring the hardware RTK, we still need to address observations of the software at abstraction levels other than just the process-level. For instance, how can we track a software process' function-call hierarchy, or how to monitor data variables, or the execution of a particular instruction? In certain cases it might also be necessary to sample register contents in a CPU, an act which is not obvious without software instrumentation, or special hardware support in the CPU. Employing a hardware monitor which passively listens to a CPU's address and data buses may be inadequate, or even useless, if the CPU is equipped with an instruction and/or data cache - which today is more or less typical rather than exceptional. Therefore, with this background, it is motivated to formulate the following question:

**Question 3 (Q3).** *Is it possible to monitor software execution and data, at any abstraction level, in a solution to both Q1 and Q2?*

The answer to Q3 is also given in Contribution A and B, and Contributions C and D respectively provides a validation of that answer.



---

## Chapter 4

# Contributions

---

In this chapter we will briefly describe the main contributions presented in this thesis.

***Contribution A: Concept of a Uniform Hardware/Software Monitor***

Our central contribution is the concept of a monitoring system that can be applied for observations of a computer system's hardware and software components. This monitoring system, designated *Multipurpose/Multiprocessor Application Monitor* (MAMon), is based on an integrated hardware probe unit (IPU) which is integrated with the observed system's hardware. The IPU collects events of interest in the system, and transfers them out of the system to a dedicated computer where the events can be analysed without perturbing the observed system's behaviour. In the case where the observed system incorporates a hardware RTK component, the IPU may also be connected to that component's internal signals and data structures in order to extract process-level information.

The main advantages with the MAMon concept are:

- a) detection and collection of events occurs non-intrusively to the system, or with a minimum of impact should the software require instrumentation for hybrid monitoring,
- b) hardware and software events are monitored using the same device – i.e. uniform monitoring is achieved – and are displayed and analysed using the same monitor applications and tools, and

- c) the IPU may be implemented as an IP-component for integration in a SoC, thus overcoming the difficulties related to probing obstructive hardware.

In Paper A we introduce the MAMon concept from a verification of SoCs perspective, and in Paper B the ideas are refined for more general applicability. Paper B also describes the integration with the RTK in more detail, and gives an overall architectural description of the monitoring system.

Contributions B through D presented below are validations of the MAMon concept for various system configurations and applications. It should be mentioned that MAMon has been applied in a HW/SW co-simulation model of a SoC comprising a CPU (an ISS-model of a PowerPC 60x), an RTK, and the monitor's IPU. However, this configuration is not documented or validated thoroughly, hence, it is not listed as a contribution of its own.

#### ***Contribution B: A Monitor for a Multiprocessor System***

The second contribution, presented in Paper B, is an implementation of MAMon for a real-time multiprocessor based on *commercial-off-the-shelf* (COTS) hardware. This multiprocessor system is a research platform built for studies on hardware-acceleration for RTOSs [LKF99, KL99]. Our aim is to build a monitor that is able to observe the behaviour of multiprocessing software run by a hardware-accelerated RTOS, i.e. a hardware RTK.

The implementation resulted in a hardware prototype based on a FPGA (a Xilinx Virtex-1000) that is configured with the hardware RTK and the monitor's IPU. Using the monitor in combination with the RTK we are able to analyse the software's behaviour at the task-level, running on up to three CPUs. The analysis is done with no intrusion on the system's behaviour or timing. With an addition of instrumentation of the software it is also possible to utilise the monitor in a hybrid manner, with a cost per *instrumentation probe* reduced to the time-length of a 32-bit bus write cycle (in this case a PCI-bus@33 MHz).

In Paper B we present the full details of the implementation, and describe the tools we also developed to control the monitoring process and analyse collected data (see Contribution E below).

#### ***Contribution C: A Monitor for a Single-Processor System***

In another validation of our monitor we were interested in studying the performance differences between a single-processor system running a hardware-accelerated RTOS [Fur00] and a software-only RTOS, called *SW Symo* [Riz01]. The idea was to compare the amount of idle execution time (i.e. when no tasks

are running) for the same software when run on each system, an experiment which would reveal the execution overhead imposed by the RTOS. The experiment involved adaptation of the hardware IPU – used in the multiprocessor system in Contribution B – with the ability to detect currently executing tasks managed by the SW Symo RTOS. The task id:s are extracted using instrumentation of the SW Symo’s context-switch routine, so that the currently active task’s id is written to a memory-mapped register in the IPU (i.e. an IPU software probe register).

The experiment was part of a master thesis project, carried out by Al-Wandi and Hessadi [AWH02], under supervision of the author of this thesis. They also designed two graphical interface tools that visualise CPU work load in the studied systems; one tool to visualise *live* CPU work load (i.e. CPU utilization), and the other tool to show *historical* CPU load (see Contribution E below). Due to their limited project time they managed, however, only to finish the experiments with the SW Symo target system.

#### ***Contribution D: A Hybrid Monitor for Cache Performance Analysis***

In [Seb02a], Sebek used the hybrid monitoring feature of the monitor (described in Contribution B) in order to analyse cache behaviour in a real-time system. Sebek’s objective was to measure the *execution delay* that relates from task pre-emption in a multitasking single-processor system [SG02, Seb02b]. Using the built-in performance monitors in a MPC750 CPU [Mot97], he was able to construct software which reads cache-related performance properties, and by using the monitor’s time-stamp function he also measured execution times in order to determine the *cache-related pre-emption delay* (CRPD) as well as the *threshold miss-ratio* values for an instruction cache. Thus, his solution to minimise software overhead was to write the cache-related data to dedicated *software probe registers* in the IPU. The IPU collected the data, and packaged it into "software probe" events which were time-stamped and sent further to an external PC where the data was analysed.

Using our monitor, Sebek was able to measure very accurately, and analyse, the CRPD in a real-time system.

#### ***Contribution E: A Framework for Monitoring Applications and Tools***

To make use of the collected events from the monitor, e.g. for analysis and visualisation purposes, we developed an application platform that enables easy and rapid design of customised monitor tools (see Paper B and Paper C). This platform, which is based on a modular design implemented using the JAVA object-oriented language, provides support for communication with the IPU

(i.e. hardware driver), a relational (SQL) database for structured and well-defined storage/access of the collected events, and a GUI environment for user interaction with the monitor and visualisation of collected events. Initially, the platform constituted only a tool to graphically depict events collected from a SoC comprising one CPU and a RTK (described in Paper A). In the following work, presented in Paper B, the platform was developed to become a framework with more general applicability.

The usability of the framework was validated also in the work by Al-Wandi and Hessadi [AWH02] who implemented tools that visualise CPU work load (see Contribution C). We have also extended the framework with support for USB-communication with the IPU. This latter effort was carried out by Andreas Malmquist [Mal04] under supervision of the author of this thesis.

In Paper C we give a more comprehensive documentation of the framework, its components, and the tools it currently supports.

#### ***Contribution F: Patent***

Our final contribution is a patent on the MAMon concept described in contributions A through D. The patent, which currently is registered in Sweden (patent no **SE517917**), was acquired by *RealFast AB*; a company specialised in developing IP-components for the FPGA and SoC market. Their motivation for acquiring and exploiting a patent based on our ideas shows an industrial relevance, and an interest in our work. A valid and registered patent also gives a proof of uniqueness since it has been reviewed by patent engineers and patent registration authorities.

The patent application was authored in co-operation with *Ann-Marie Reyier* at *Bjerkéns Patentbyrå* in Västerås, Sweden. Mrs Reyier wrote the application based on our documentation and pre-published papers. For reference, we have included the patent description and its claims in Appendix A.

---

## Chapter 5

# Summary of Papers

---

### 5.1 *Summary of Paper A*

Mohammed El Shobaki and Lennart Lindh, A Hardware and Software Monitor for High-Level System-on-Chip Verification, In proceedings of the IEEE International Symposium on Quality Electronic Design, San Jose, CA, USA, March 2001.

**Summary:** The paper describes our concept of an *on-chip hardware monitor* for uniform monitoring of hardware/software systems-on-chip (SoC). For hardware analysis the monitor detects and collects events at the register-transfer level (RTL), performing very much like a logic analyser. For software analysis the monitor may be attached to processing elements in the SoC, e.g. to processor interconnects and buses, in order to extract software instructions and data. In this latter sense the monitor works non-intrusively to the system, or with a minimum of interference if the software is instrumented for hybrid monitoring. In the paper we also relate to our previous work on hardware-implemented Real-Time Kernels, and discuss how such an implementation may be integrated with the monitor in order to extract process-level events without perturbing a system's functional, timing, and performance behaviour. This property is required especially when debugging and analysing SoCs used in real-time systems. We also motivate the use of the monitor in a top-down debugging strategy where it can be employed in the early stages of verification and validation at a system or process level, and later, at the hardware's RTL whenever more level of detail is requested. Finally, the paper describes the embryo to a full-

featured monitoring application framework with support for monitor control, event tracing, and visualisation of performance and data.

**My contribution:** The paper is written by me, under supervision of Lennart Lindh.

## 5.2 Summary of Paper B

Mohammed El Shobaki, On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems, In proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA), Tokyo, Japan, March 2002.

**Summary:** The paper presents further developments to the concepts discussed in Paper A, and describes a physical implementation of a prototype monitor along with a much more developed version of the framework for monitoring applications. The monitor hardware is realised in a *probe unit* (IPU) that is integrated with a hardware RTK in an FPGA-chip, which in turn is mounted on a PCI-board in a *commercial-off-the-shelf* multiprocessor system. In this setup the RTK manages real-time process scheduling for up to three CPU-boards hosting PowerPC 60x/75x processors. The monitor, which has probes tightly coupled to the RTK's data paths, logs all scheduling events in the RTK, as well as its other features, e.g. inter-process communication events, semaphore state transition events, external interrupts, etc. The logged events are time-stamped and then transferred through a dedicated connection to an external PC, where the events are stored in a relational database, ready to be accessed by monitoring applications.

The software that access collected events, or controls the monitor, are implemented as separate modules which are plugged into a GUI-platform developed using Java. The paper describes the architecture of this GUI-platform, which we choose to call a *framework* since it is designed to be easily customisable and upgradeable.

Moreover, the paper demonstrates the use of the monitoring system and presents some implementation data. The paper's main conclusion is that it is possible to non-interferingly observe the behaviour of software processes by monitoring the hardware RTK.

**My contribution:** I'm the sole author of the paper.

### 5.3 *Summary of Paper C*

Mohammed El Shobaki and Jeroen Heijmans, MAMon - A Multipurpose Application Monitor, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-121/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden, September 2004.

**Summary:** This report describes the architecture and integral components of the Multipurpose Application Monitor (MAMon). The IPU's signal interface is described and we explain how the interface conforms to integration with a hardware RTK and the communication port to an external host computer. The report also provides a programmer guide for the monitoring application framework, as well as a user manual for the currently implemented monitoring tools within the framework. The report is mainly intended as a reference guide for working with MAMon at a user's level.

**My contribution:** I compiled this paper based on several documents, partly written by me and partly by my co-author, Mr Heijmans. Mr Heijmans contributed to the implementation of MAMon's host software, an effort which he conducted in a special student project under my supervision.





---

## Chapter 6

# Related Work

---

This chapter presents academic research and industrial practice that relate to our work. We will first give an overview of the related work, then present some of the surveyed systems more in detail, and give a discussion on their relation to our contributions. For more extensive surveys on monitoring systems see [TBHS96] and [Sch94].

First, we must emphasize that MAMon is primarily a process-level monitor for systems employing a hardware RTK, and secondary it is a multipurpose hybrid monitor in that it can also be used to record software-generated events. These two features may be combined in an embodiment of the monitor, but may also be used separately. For simplicity we differentiate between the two features in the following discussions. Section 6.1 presents related work on monitoring RTKs, Section 6.2 describe work on hardware-only monitoring systems, and Section 6.3 presents work on hybrid monitoring systems. Finally, in Section 6.4 we relate our contributions to on-chip techniques that has emerged with the developments towards hardware/software co-design and systems-on-chip.

### *6.1 Monitoring Real-Time Kernels*

To our knowledge, our research on monitoring hardware-implemented RTKs is unique. We believe this is mainly due to limited research efforts on hardware RTKs. However, for software-implemented RTKs there exists a number of proposed systems. For a few examples see [TKM89, TFC90a, Win04, Men98b]

***The ARTS monitor***

In [TKM89], Tokuda et al. presents a research prototype on a real-time monitor developed for the ARTS distributed operating system. The proposed monitor requires support from ARTS kernel such as notification of process' state-changes, e.g. process creation, waking-up from a blocked state, being scheduled, etc. The events collected from the target system are sent to a remote host for display of the execution history.

***The "Imitating" Monitor***

The work by Tsai et al. in [TFC90b, TFC90a] presents a more generic real-time hardware monitor which is suitable for observations at various abstraction levels, including the *process-level*. This monitor is realised as a separate computer system that is attached to the buses of the computer under observation, and utilise bus-snooping in order to reflect the observed system's state onto the monitoring system. Thus, the monitor *imitates* the execution at the observed system. We discuss this monitor further in the next section.

***Xpert Profiler***

*Xpert Profiler* by Mentor Graphics [Men98b] is a typical commercially available tool for profiling, measurement, and evaluation of software written for the VRTX RTOS [Men98a]. *Xpert Profiler* relies on software-instrumentation that may be automatically inserted in the RTOS kernel whenever task-switching activities are to be monitored. The monitoring process is controlled by a program on separate host computer (see Figure 6.1), where the recorded data is also visualised and processed. On the target side, the application is linked with the libraries (*proflib.o* and *proflib\_vrtxsa/vrtx32.o*) containing service-routines that can be called (either by the RTK or the application) at the instances where samples are requested. The sample buffer, which also resides on the target side, contains the samples that are collected during a monitoring session.

***Discussion:*** The main difference between MAMon and the above described systems is the notification of events which for the surveyed systems (except the Imitating monitor) require instrumentation of the software RTKs, while in MAMon the kernel-events are sampled directly from a hardware RTK. In case of the ARTS monitor the intrusiveness due to the software instrumentation is included in the schedulability analysis. The commercial monitors, like *Xpert Profiler* [Men98b] and its equals, do not take any measures to eliminate execution interference or probe-effects.

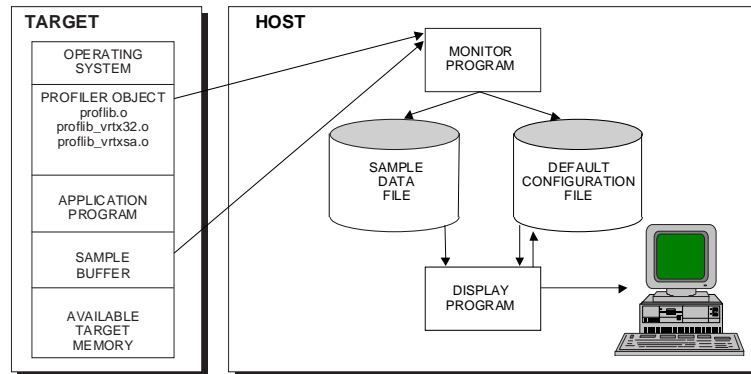


Figure 6.1: The components of Xpert Profiler [Men98b].

## 6.2 Hardware Monitoring Systems

### *The Bus-listener and The Imitating Monitor*

Research prototypes on hardware-supported monitoring for single-processor systems can be found in the work by Plattner [Pla84], and Tsai et al. [TFC90b]. Common techniques in these approaches are to employ bus-listeners to reflect memory-accesses onto a separate memory system, and the use of imitating processors, i.e. in order to perform intrusive monitoring on a (non-intrusively) reflected system. For instance, Plattner's monitor, called *bus listener*, is attached to the target processor's buses in order to detect memory transactions which then are reflected onto a separate memory designated *phantom memory*. From this memory it is then possible to examine the system's state without interfering with the observed system. Tsai et al.'s monitor is also attached to the processor buses, however, the detected signal activities are used to synchronise a separate computer node (with an own processor, memory system, and I/O-devices). This computer node imitates the observed computer, and allows for examination of the system state without interference with the monitored system.

### *The Interrupt Priority Level Monitor*

In [JC91], Baker and Crilly presents an architecture for monitoring interrupt priority levels on an interrupt-driven processor. The monitor is used to exter-

nally observe the interrupt levels at which the processor execute and the time spent at each priority level in order to determine processor and service-routine efficiency. The concept is proposed for a generic interrupt-driven microprocessor, and presumes that information on interrupt activities can be extracted off-chip via the processor bus-interface.

Figure 6.2 shows an overview of the monitoring system for a generic target processor called DUEL. The monitor hardware observes the DUEL processor for interrupts, log events, and buffer event records. The SCSI bus is used for host control of the monitor hardware, and for sending event records to the host for post-analysis. The host system controls the monitor hardware, collects event records, and reconstructs the event history for use with analysis tools.

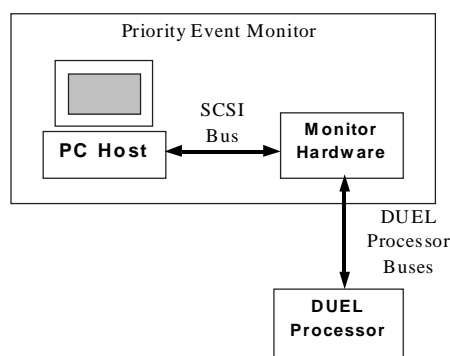


Figure 6.2: Baker and Crilly's Priority Event Monitor [JC91]

### *The Ted Monitor*

In yet another related work, Liu and Parthasarathi [LP89] presents a hardware monitor for a shared-memory multiprocessor system called *Ted* (Testbed for Distributed Processing). They propose this monitor to be used in debugging and run-time process-scheduling. In the following we will describe this monitor in detail to give a more complete presentation of a work that relates to the multiprocessor implementation of MAMon (presented in Paper B).

The Ted consists of Intel 8086 single board computers that are organised in clusters. Within a cluster, the single board computers are connected to a *Multibus* and communicate via shared memory. Clusters are connected with an Ethernet network. Figure 6.3 shows the architecture of Ted. The monitoring

hardware is attached via a probe interface to the Multibus on the cluster to be monitored. In this approach, the monitoring device detects events on the Multibus and collects the data for use by an external system. The external system may be a completely independent computer, or it could be one of the board computers (dedicated for monitoring) within the cluster as is proposed by the authors. In the latter case, the monitor is controlled via the Multibus, hence, a slight interference will occur. However, when monitored data is collected from the monitor, the parallel port of the processor board is used in order to avoid contentions on the Multibus.

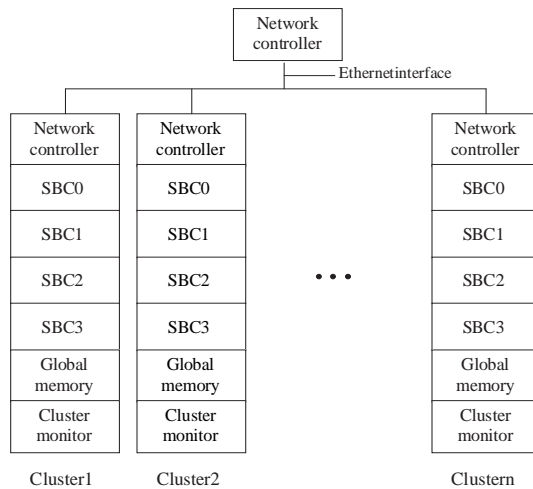


Figure 6.3: The Ted architecture [LP89]

Figure 6.4 shows the internal architecture of the monitoring device. The probes of the monitor connects to the bus within the cluster. A standard Multibus edge connector serves this function. Via the monitor’s bus interface, the bus signals are sampled and compared with a set of predefined signal patterns. This is done in the event filter where a set of user-programmable registers contains the patterns for the events of interest. The bit pattern consists of the following signal lines:

- 20-bit address lines
- 16-bit data lines, and

- 4-bit read/write lines (I/O and memory r/w)

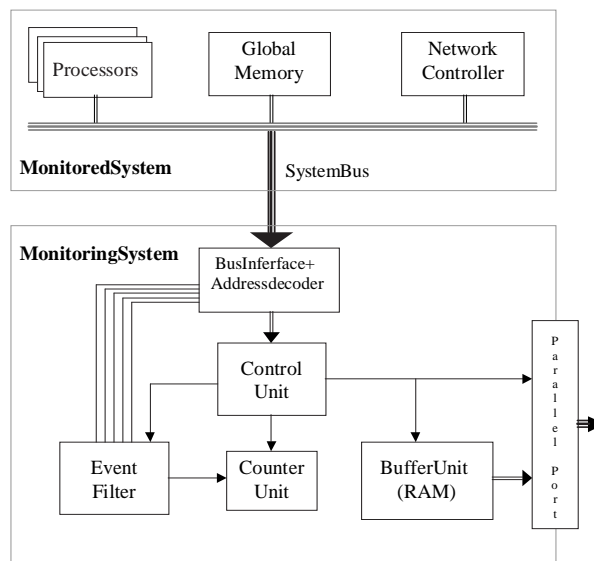


Figure 6.4: Functional block diagram of the Ted monitor [LP89]

When a match occurs, the event filter triggers a clock-counter mechanism which counts the elapsed number of clock-cycles between two consecutive events. The event data and the counter-values are then stored in a local buffer which is managed by the buffer unit. The buffer unit manages the buffer memory using a double-buffering scheme which allows one buffer to be filled while the other is emptied. When a buffer fills, the control unit starts transmitting the buffer contents to the external system via the parallel port.

**Discussion:** Compared with the above monitoring systems (except the *Priority Event Monitor*), MAMon's IPU is not a hardware monitor that collects any data present on physical wires in the system, at least it is not intended for such use in the first place. However, the IPU may easily be customised for such purposes as well. On the other hand, the monitoring systems above are obsolete in computers that employ cache-memories that obstructs off-chip probes from observing processor execution. Since the IPU may be placed in a SoC it is

possible to overcome the problem with obstructive caches by attaching the IPU directly in between – soft IP – processor cores and their caches.

When comparing MAMon with Baker and Crilly's monitor, MAMon provides the same features for observing interrupts and their priorities.

### 6.3 Hybrid Monitoring Systems

#### ***The Co-Processor "Flight Recorder" Monitor***

Research prototypes for hybrid monitoring systems have been proposed in [HW90, MSHA98, Gor91, Sch94, CP98]. The common approach in most of these systems is to employ a hardware monitor that collects instrumentation-generated events via a memory-mapped register interface. An exception to this model, however, is the monitor proposed by Gorlick [Gor91]. His monitor is implemented using a co-processor which executes special monitoring code that is inserted, manually by the user or automatically by the compiler, at the locations of interest. The monitoring process is triggered by an event occurrence in the target processor, resulting in that the co-processor starts executing the monitoring code while the target processor continues normal program execution. In this manner, the co-processor may perform the monitoring operations with a very low impact on the system's performance. Gorlick compares his monitor with the operation of a flight-recorder in that it saves traces of the execution history.

#### ***MultiKron***

The *MultiKron*<sup>1</sup> project presented in [Min94] by Mink et al. is an attempt to provide a chip implementation of a hybrid monitor. This chip, called MultiKron, supports event tracing and *performance counters* in combination, or only the performance counters in one version of the chip. The performance counters are generic 32-bit registers that may be used for counting the number of occurrences of a target event or to record the elapsed time between events. In [MSHA98] they present a prototype implementation for inclusion in computer systems with a PCI-bus [PCI]. In the same paper they also demonstrate the integration of the monitor with a tool suite for parallel performance measurements called Paradyn [MCC<sup>+</sup>95].

---

<sup>1</sup>MultiKron is a registered trademark of the National Inst. Of Standards and Tech. (NIST), USA

***Transprobe***

Yet another chip implementation of a hybrid monitor is the *Transprobe* proposed by Calvez and Pasquier [CP98]. Their monitor is also a hardware monitor in that it also provides a mechanism to latch in arbitrary signals in the hardware. They also advocate using the monitor in an on-chip solution to overcome obstructive hardware packaging, and reach in deeply behind cache memories. To monitor distributed system, i.e. multiple computer nodes, the *Transprobe* may be connected with other *Transprobe* chips using a custom token-ring serial network called *Transbus*. The *Transbus* operates with an effective bandwidth of 1 Mbyte/s, thus allowing an event throughput of up to 100,000 events/s (each event packet is 10 bytes). They also provide an in-house developed tool suite that supports various performance assessments.

***Discussion:*** MAMon provides hybrid monitoring in the same way as the majority of the surveyed monitors above. MAMon is however more similar to the monitor proposed by Calvez and Pasquier as their implementation of the *Transprobe* chip should also allow for incorporation in a SoC. However, in the current implementations of MAMon there may only exist one IPU in the observed system.

## 6.4 On-Chip Techniques

With the developments towards increased integration of hardware and software components and utilisation of hardware implementation technologies, it is becoming more and more attractive to put complete systems into single chip solutions, i.e. systems-on-chip that incorporates one or more processor cores, custom IP, I/O control components, and sometimes even reconfigurable hardware block. To support the development of these type of systems we see increased efforts to provide debugging capabilities right on the SoC [YS99, HL99, Chi00, Sig, Sca].

***ARM's EmbeddedICE***

The EmbeddedICE Logic [YS99, Lim99] (see Figure 6.5) provides on-chip debugging capabilities for the *ARM* series of processor cores. This module comes as an integral part of an ARM-processor, and supports breakpointing and stepping code, non-intrusive monitoring of instruction flow and data access, and the ability to modify memory without halting the processor, and all this during full-speed execution. The module is controlled from a debugger – running on



a separate host computer – through a standard JTAG port [JTA93]. Due to the tight coupling with ARM’s processor architectures, the EmbeddedICE module can provide these features seamlessly.

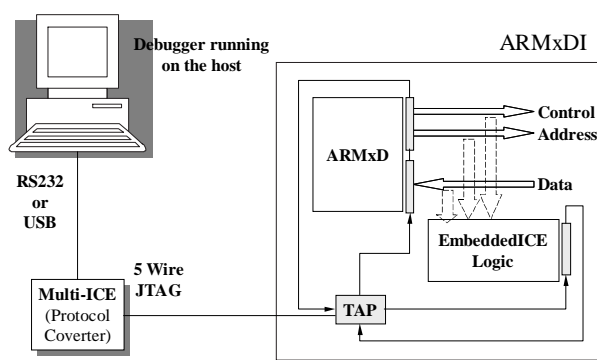


Figure 6.5: ARM’s EmbeddedICE Logic for On-Chip Debugging [YS99]

### **ChipScope and SignalTap**

The *ChipScope ILA* (Integrated Logic Analyzer, by *Xilinx Inc.* [Chi00]) is another on-chip solution for analysing logic in Xilinx’ reconfigurable hardware chips. The approach is to integrate a logic analyser IP-component on-board Xilinx FPGAs. The ILA component, which is connected in HDL and then synthesised with the design, collects logic-level events in run-time, and transfers the events via the JTAG port over to a host computer where the data is presented in a waveform graph. Thus, the ILA gives the designer full access to all internal nodes and the (FPGA’s) data bus at full system speeds. A similar solution is also provided by *Altera Inc.* [Sig] in their *SignalTap II* embedded logic analyzer.

**Discussion:** The *event trace* feature of the EmbeddedICE Logic module for ARM processors may just as MAMon be used for hardware and hybrid monitoring of software execution. However, this module is not generally applicable with other processors.

To compare MAMon and its IPU with the other on-chip techniques describe above is not straightforward, since they do not address the same questions we formulated in Chapter 3. However, the hardware/software system development

support they provide altogether indicates the need for increased observability in today's state-of-the-art computer systems, a trend which strongly confirms our overall motivations for this work.

---

## Chapter 7

# Conclusions

---

The main conclusions from this thesis are summarised below:

- We have proposed a monitoring concept, designated *MAMon*, and successfully proved its use in a number of implementations. *MAMon* offers non-interference observability of computer systems that employ a hardware-accelerated RTOS. The central contribution is the concept of an *integrated hardware probe unit* (IPU) which passively detects and collects process-level information from the hardware-accelerated RTOS, and communicates this data to an external computer for analysis, debug, and visualisation.
- We have also proved it possible to utilise *MAMon* for combined observations of hardware and software behaviour. In the latter, the IPU is used to trace software execution and data through instrumentation of the software. To avoid probe-effects in concurrent programs we suggest the instrumentation to be kept in the deployed product.
- It is also shown in a case-study that *MAMon* may be used to measure task's execution time and cache performance behaviour [Seb02a]. From this study we conclude that task execution properties may be measured very precisely during run-time and full-speed execution. Another advantage is that *MAMon* may be employed in a real execution environment, where the observed system is exposed to natural behaviour and disturbances from the environment. These properties of the monitor makes it very useful for task execution and resource utilisation analysis that is done prior to the scheduling process of real-time computer software.

- The monitor's IPU has a small hardware footprint which makes it attractive to integrate in systems comprising custom hardware, such as FPGAs and ASICs. As hardware implementation technologies are becoming cheaper (e.g. FPGAs and CPLDs), our proposed monitor delivers observability at a low cost.

Despite that MAMon initially was tailored to observe the behaviour of the hardware-accelerated RTOS, we believe that it is quite general and may be explored further to also address testing and analysis of hardware, and Systems-on-Chip in particular. The fact that the IPU may be implemented as an *intellectual property* component (IP) for integration in a SoC make it interesting and relevant for use in industrial practice. This is also confirmed by the support for patenting the concept.

Furthermore, we conclude that there are disadvantages with the MAMon concept regarding its dependability on a communication channel that must provide - or by other means be capable of delivering - performance that is proportional to the amount of generated run-time information in the system. This sort of "*Achilles heel*" of MAMon is in fact dependent on several properties: the type of application that is monitored, the monitored level of abstraction (the lower, the more information), the size of the IPU's FIFO-buffer, the bandwidth of the communication channel and its buffering requirements on the host peer, and the performance of file operations and database storage at the host computer system. The FIFO-buffer in the IPU must be dimensioned to handle the event input rate with respect to the type of application monitored, and the bandwidth of the communication channel. As a precaution measure we have, however, provided a way of indicating FIFO overruns in the IPU, so that we can determine *if* and *when* the FIFO buffer-size is underestimated.

---

## Chapter 8

# Future Work

---

Current trends in hardware design shows an increasing use and re-use of IP components. The components may either be standardised such as those included in design tool libraries or frameworks, or may come from third-party vendors who provide specialised components. This development towards reusability resembles very much recent developments in the *software engineering* approach to software design. As in software engineering, hardware designers face an increasing complexity in the verification and validation process of large SoCs that comprise components originating from different sources/vendors, and components with different versions and configurations. Although the components are verified standalone, it may be difficult to predict their run-time behaviour when they are compound in a SoC. To manage product platforms based on such premises, and handle evolutionary system changes is an increasing challenge for SoC developers. Therefore it is motivated to find methods and develop tools that facilitates several aspects in SoC development.

We believe the MAMon approach with on-chip IPU-components can be utilised better for run-time observations of components within SoC, not only for development purposes, but also in deployed products with complex behaviour. The underlying idea is to have on-board monitoring facilities throughout the lifetime of SoC-based products, in particular such products that have safety critical, or high quality, requirements. Just as a "flight-recorder" [Gor91], the recorded traces from the monitor may be used as input to post-crash analysis tools and debuggers [TSHPO3]. The traces should be long enough to cover faulty execution histories, and be kept in non-volatile memory embedded in the system (on-chip or off-chip the SoC). These ideas are already implemented

for traditional board computer systems. However, we believe there will be an increased demand for this type of facilities in SoC-based systems.

*Hardware/software co-design* is another research field where monitoring may come to play an important role. The main objectives of hardware/software co-design is to raise the design abstraction to a behavioural description level [SW97, MS96, FN01], and optimise a system design by partitioning its implementation into hardware and software technology respectively. The partitioning may be optimised with respect to several parameters, e.g., execution performance, chip area costs and pin constraints, power consumption, and trade-offs between the parameters. Problems that relate to hw/sw co-design are the difficulties in debugging a design that partly resides in hardware, and partly in software. Hence, a co-debugging strategy must be adopted. Koch et al. [KRK98] propose interactive breakpoint-based debugging using co-emulation of hw/sw systems. Their approach provides the ability to set breakpoints in either technology, and synchronise hardware and software processes that halts due to breakpoints. Potkonjak et al. [KP97, KKH<sup>+</sup>01] propose an approach to symbolic debugging of behavioural descriptions. Their efforts are motivated by the need to synchronise variable values in the hw/sw implementation with their related symbols in the behavioural description.

To facilitate co-designed system development we believe a monitoring approach like MAMon could be utilised for run-time observations. An experimental idea is to extend hw/sw co-synthesis with generation of instrumentation for processes in software, and generate customised IPUs that are attached with the synthesised hardware.

---

# Bibliography

---

- [AFLS96] Joakim Adomat, Johan Furunäs, Lennart Lindh, and Johan Stärner. Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, June 1996.
- [AWH02] Adil Al-Wandi and Mehrdad Hessadi. Monitoring of System Work Load, Master Thesis Report, Västerås, Sweden, May 2002.
- [Baw99] H. Bawtree. Real-time monitoring with StethoScope 5.1. *Software Development*, 7(9), September 1999.
- [But97] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.
- [Chi00] Chipscope integrated logic analyzer. Xilinx Inc., San Jose, CA 95124-3400, June 2000. <http://www.xilinx.com/products/chipscope/>.
- [CP98] J.P. Calvez and O. Pasquier. Performance monitoring and assessment of embedded hw/sw systems. *Design Automation for Embedded Systems journal, Kluwer A.P.*, 3:5–22, 1998.
- [FN01] Masahiro Fujita and Hiroshi Nakamura. The standard SpecC language. In *ISSS*, pages 81–86, 2001.
- [Fur00] Johan Furunäs. Benchmarking of a real-time system that utilises a booster, 2000.
- [Gai86] Jason Gait. A probe effect in concurrent programs. *Software - Practise and Experience*, 16(3):225–233, March 1986.

- [Gla80] Robert L. Glass. Real-time: The "lost world" of software debugging and testing. *Communications of the ACM*, 23(5):264–271, May 1980.
- [Gor91] Michael M. Gorlick. The flight recorder: An architectural aid for system monitoring. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 175–182, Santa Cruz, CA, USA, May 1991.
- [HL99] Ing-Jer Huang and Tai-An Lu. ICEBERG: an embedded in-circuit emulator synthesizer for microcontrollers. In *Proceedings of the Design Automation Conference*, pages 580–585, June 1999.
- [HW90] Dieter Haban and Dieter Wybraniec. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Trans. on Software Engineering*, 16(2):197–211, February 1990.
- [JC91] Kenneth F. Baker Jr. and Paul Benjamin Crilly. A priority event monitor for an interrupt-driven microprocessor. In *Proceedings of Southeastcon '91*, volume 2, pages 905–909. IEEE, April 1991.
- [JRR94] Farnam Jahanian, Ragunathan Rajkumar, and Sitaram C.V. Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–273, November 1994.
- [JTA93] JTAG. Ieee standard test access port and boundary-scan architecture, 1993.
- [KKH<sup>+</sup>01] Farinaz Koushanfar, Darko Kirovski, Inki Hong, Miodrag Potkonjak, and Marios C. Papaefthymiou. Symbolic debugging of embedded hardware and software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):392–401, March 2001.
- [KL99] Tommy Klevin and Lennart Lindh. Scalable architectures for real-time applications and use of bus-monitoring. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, China, December 1999.



- [KP97] Darko Kirovski and Miodrag Potkonjak. A quantitative approach to functional debugging. In *ICCAD*, pages 170–173, 1997.
- [KRK98] Gernot H. Koch, W. Rosenstiel, and U. Kebschull. Breakpoints and breakpoint detection in source-level emulation. *ACM Transactions on Design Automation of Electronic Systems.*, 3(2):209–230, 1998.
- [Lab02] Jean J. Labrosse. *MicroC OS II: The Real Time Kernel*. CMP Books, CMP Media LLC, 2002.
- [Lim99] ARM Limited. *Application Note 31, Using EmbeddedICE*, February 1999.
- [Lin92] Lennart Lindh. FASTHARD – A Fast Time Deterministic Hardware Based Real-Time Kernel. In *Proceedings of the Fourth Euromicro Workshop on Real-Time Systems*, pages 21–25, June 1992.
- [Liu00] Jane W. S. Liu. *Real-Time Systems*. Pearson US Imports & PHIPES, 2000.
- [LKF99] Lennart Lindh, Tommy Klevin, and Johan Furunäs. Scalable architectures for real-time applications - SARA. In *CAD & CG'99*, December 1999.
- [LP89] An-Chi Liu and Ranjani Parthasarathi. Hardware monitoring of a multiprocessor system. *IEEE Micro*, pages 44–51, October 1989.
- [LSF<sup>+</sup>98] Lennart Lindh, Johan Stärner, Johan Furunäs, Joakim Adomat, and Mohammed El Shobaki. Hardware accelerator for single and multiprocessor real-time operating systems. In *Proceedings of the Seventh Swedish Workshop on Computer Systems Architecture*, Göteborg, Sweden, June 1998.
- [Mal04] Andreas Malmquist. MAMON 1.1 USB – IMPLEMENTATION REPORT. Technical report, Department of Computer Science and Engineering, April 2004.
- [MCC<sup>+</sup>95] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel

- performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
- [Men98a] Mentor Graphics, Microtec Division. *VRTX Real-Time Operating System*, 1998.
- [Men98b] Mentor Graphics, Microtec Division. *XPERT Profiler - Measurement and Evaluation Tool*, 1998.
- [MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–621, December 1989.
- [Min94] A. Mink. Operating principles of multikron ii performance instrumentation for mimd computers, 1994.
- [Mot97] Motorola Corp. *MPC750 RISC Microprocessor Users Manual*, August 1997.
- [MRS<sup>+</sup>90] L. D. Molesky, Krithi Ramamritham, C. Shen, J. A. Stankovic, and G. Zlokapa. Implementing a predictable real-time multiprocessor kernel - the spring kernel. In *IEEE Workshop on Real-Time Operating Systems and Software*, May 1990.
- [MS96] G. De Micheli and M. Sami. Hardware-software codesign, 1996.
- [MSHA98] Alan Mink, Wayne Salamon, Jeffrey K. Hollingsworth, and Ramu Arunachalam. Performance measurement using low perturbation and high precision hardware assists. In *RTSS*, pages 379–, 1998.
- [NSMT<sup>+</sup>00] Christer Norström, Kristian Sandström, Jukka Maki-Turja, Hans Hansson, Henrik Thane, and Jan Gustafsson. *Robusta Real-tidssystem*. MRTC Mälardalen Real-Time Research Centre, 2000.
- [NUI<sup>+</sup>95] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In *Proceedings of TRON'95*, pages 34–42, 1995.
- [PCI] PCI Industrial Computers Manufacturers Group. *CompactPCI Specification rev. 2.1*.

- [Pla84] B. Plattner. Real-time execution monitoring. *IEEE Trans. on Software Engineering*, SE-10(6):756–764, November 1984.
- [Riz01] Larisa Rizvanovic. Comparison between Real time Operative systems in hardware and software, Master Thesis Report, Västerås, Sweden, September 2001.
- [Sca] Scanner module – trace 32 technical information. Lauterbach Datentechnik GmbH, Fichenstr. 27, D-85649 Hofolding. <http://www.lauterbach.com/>.
- [Sch94] Ulrich Schmid. Monitoring distributed real-time systems. *Real-Time Systems*, 7(1):33–56, July 1994.
- [Seb02a] Filip Sebek. Instruction cache memory issues in real-time systems. Licentiate thesis 02/60, Mälardalen Real-Time Research Centre, Department of Computer Science and Engineering, Mälardalen University, Sweden, October 11 2002.
- [Seb02b] Filip Sebek. The real cost of task pre-emptions — measuring real-time-related cache performance with a hw/sw hybrid technique. Technical Report 02/58, Mälardalen Real-Time Research Centre, August 2002.
- [SG02] Filip Sebek and Jan Gustafsson. Determining the worst-case instruction cache miss-ratio. In *Proceedings of ESCODES 2002*, San José, CA, USA, September 2002.
- [Sig] SignalTap II, Altera Corporation. <http://www.altera.com/>. 101 Innovation Drive, San Jose, CA 95134.
- [SVS<sup>+</sup>88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin. Fault injection based automated testing environment. In *Proceedings of the 18th Intl. Symposium on Fault-Tolerant Computing (FTCS18)*, pages 102–107, Tokyo, Japan, 1988.
- [SW97] Jorgen Staunstrup and Wayne Wolf. *Hardware/Software Co-Design, Principles and Practice*. Kluwer Academic Publishers, 1997.

- [TBHS96] Jeffrey J.P. Tsai, Yaodong Bi, Steve J.H. Hang, and Ross A.W. Smith. *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis*. Wiley-Interscience, 1996.
- [TFC90a] Jeffrey J.P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen. A non-interference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Trans. on Software Engineering*, SE-16(8):897–916, August 1990.
- [TFC90b] Jeffrey J.P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen. A non-invasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, March 1990.
- [Tha99] Henrik Thane. Design for deterministic monitoring of distributed real-time systems. Technical report, Malardalen Real-Time Research Centre, Vasteras, Sweden, November 1999.
- [TKM89] H. Tokuda, M. Koreta, and C.W. Mercer. A real-time monitor for a distributed real-time operating system. *ACM Sigplan Notices*, 24(1):68–77, January 1989.
- [TSHP03] Henrik Thane, Daniel Sundmark, Joel G Huselius, and Anders Pettersson. Replay debugging of real-time systems using time machines. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 288–295, Nice, France, April 2003.
- [Win04] Wind River Systems. *WindView. TORNADO II Integrated Development Environment*, 2004.
- [YS99] Richard York and John Sharp. Real-time debug for systems-on-chip devices, June 1999.

## **II**

# **Included Papers**



---

## Chapter 9

# **Paper A: A Hardware and Software Monitor for High-Level System-on-Chip Verification**

---

Mohammed El Shobaki and Lennart Lindh  
In proceedings of the IEEE International Symposium on Quality Electronic  
Design, San Jose, CA, USA, March 2001.

### **Abstract**

Verification of today's Systems-on-Chip (SoC) occur at low abstraction-levels, typically at register-transfer level (RTL). As the complexity of SoC designs grows, it is increasingly important to move verification to higher abstraction-levels. Hardware/software co-simulation is a step in this direction, but is not sufficient due to inaccurate processor models, and slow hardware simulation speeds. System-level monitoring, commonly used for event-based software debugging, provides information about task scheduling events, inter-task communication and synchronisation, semaphores/resources, I/O interrupts, etc.

We present MAMon<sup>1</sup>, a monitoring system that can both monitor the logic-level and the system-level in single/multiprocessor SoCs. A small hardware probe-unit is integrated in the SoC design and connects via a parallel-port link to a host-based monitoring tool environment. The probe-unit collects all events in the target system in run-time, and timestamps them with a resolution of 1  $\mu$ s. The events are then stored in a database on the host for further processing. The paper will describe MAMon and how it works for software and hardware monitoring. The paper also describe how system-level monitoring can be achieved non-intrusively by using a hardware-based Real-Time Kernel.

---

<sup>1</sup>Multipurpose/Multiprocessor Application Monitor



## 9.1 Introduction

Already today many System-on-Chip (SoC) applications are hard to verify and optimise. The complexity is increasing and to verify a whole system using computer model simulations is time consuming, and some times impossible due to inaccurate models. It is also difficult to model the real-world (i.e. the environment) around the SoC.

Often a bug in a system is traced from a high-level view of the system, commonly referred to as the *system-level*, down to the *register-transfer level* (RTL) (or even lower levels if things are really bad). In this paper we refer to the system-level as to denote both the process/task-level information in software, and the behavioural-level information in hardware. As Figure 9.1 illustrates, the number of events occurring in a system are fewer at the system-level, which motivates a top-down debugging strategy.

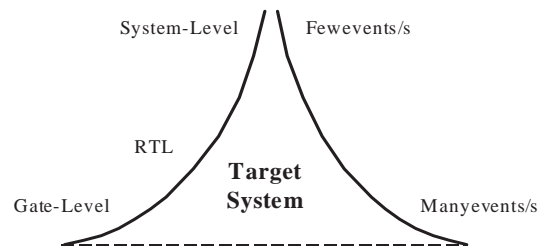


Figure 9.1: Events in a target system

Observability into SoC designs is today mostly supported for RTL verification. As SoC designs tend to increase in size and complexity, the verification process need also to take place at the system-level. Support for system-level verification already exist for the software part of a SoC, where a common technique is the use of *software monitors*. These monitors are typically found in RTOS (Real-Time Operating System) development tool environments ([Men98, Baw99]), and provides the developer with process-level information such as task-scheduling events (start/stop, block/resume, taskswitch, etc.), inter-process communication events (e.g. send and receive messages), synchronisation and resource utilization (CPUs, semaphores, I/O), external interrupts, etc. For this information to be extracted it is often required to instrument the software with special monitor instructions or processes, which later can be removed after the validation phase. The drawbacks of instrumentation is that it

utilizes target resources such as memory space and execution time from the CPUs. Also, when the instrumentation is removed, the system may change behaviour due to timing differences. This problem, commonly referred to as the *probe effect* ([Gai86, MH89]) causes many timing- and synchronisation-related errors in concurrent/distributed systems and real-time systems [TBHS96].

In this paper we present MAMon, a hardware-based monitoring system that makes a SoC observable at different abstraction-levels both in hardware and software. MAMon is a solution that integrates a small hardware component into the SoC. It works like a probe, either by listening to logic- or system-level events in a passive manner, or by being activated by software that writes to a specific register. Detected events are time-stamped and sent via a link to a host-based tool environment where the events are stored in a database. The tool environment includes a set of facilities to view, search, and analyse the events in the database. Depending on the capacity of the database disk system, and the rate at which events occur, a monitoring session (or execution history) may be several days long.

In systems running a software RTOS, the process-level events can be extracted by instrumenting the code with *hook routines* that writes information to the MAMon hardware. Hook routines are small functions that can be attached to the RTOS functions, e.g. task scheduling events, system-calls, and so on. Except the extra recourses that the instrumentation would require, it may also render problems if it has to be removed after system validation. An alternative could be to monitor the RTOS via the CPU's bus activities. This would however require that bus activities be visible outside the CPU, so cache-memories may need to be turned off if the CPU does not support monitoring of internal bus logic.

When using a hardware-based RTOS kernel [AFLS96] for the management of SoC software, we will show how the process-level events can be monitored without software overhead and with no intrusion on the system's timing behaviour. Moreover, since it is non-intrusive, it gives the SoC developer the free choice of either keeping or removing the monitor component in the final product.

The use of MAMon may be different for FPGA SoCs and ASIC SoCs. Leading FPGA manufacturers (Xilinx [Xil] and Altera [Alt]) are now promising SoC solutions with embedded *soft* or *hard* processor cores (IP). In such FPGA-based SoCs, MAMon's probe component could be instantiated and connected to the RTL and system-level description of the design during the verification and validation phase. After synthesis for the FPGA, the connections to the design are easy to change, and re-synthesise. For ASIC implementations,

however, it is much more costly to redesign. In this case MAMon may be more suitable for long-term usage as a monitor for CPU busses, and hardware-based real-time kernel events.

The paper is organised as follows: Section 9.2 gives an overview of MAMon and how it is used for hardware and software monitoring respectively. Moreover, we give a detailed description of the internal layout of the hardware, and the host interface to the tool environment. The tool environment is only described slightly. Finally, section 9.4 and 9.5 summarizes the paper with a discussion on further work and some concluding remarks.

## 9.2 MAMon

The proposed monitoring system, called MAMon, aims at providing means for event-based hardware and software debugging of single- and multiprocessor SoCs. An overview of MAMon is depicted in Figure 9.2. In this approach the monitoring system is comprised of three parts; (i) the *Probe Unit* which connects to internal SoC logic, (ii) a *Tool environment* residing on a host computer system, and (iii) the communication link between the Probe Unit and the Tool environment, called the *Host Interface*.

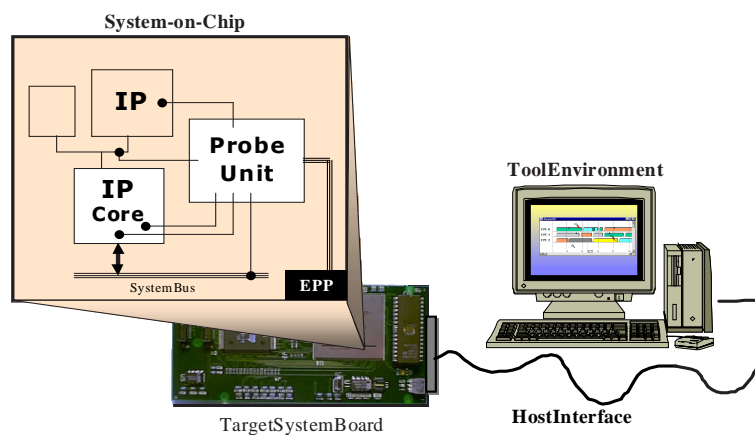


Figure 9.2: Overview of MAMon

The Probe Unit (PU, described in next section) is integrated in the design HDL code, and connected to signals that constitute the events to be monitored. For instance, an event could be defined as an access of a SoC component (IP), a certain condition on a bus (address, data, or control), arrival/contents of communication data, or an interrupt assertion, and so on. Then, in run-time, the PU performs detection, timestamping, and recording of events. Recorded events are transferred, via the Host Interface, to the Tool environment, where a database is used to store the events for further processing in display and analysis tools (section 9.2.3).

In certain cases there is need to cause events from software, for instance to monitor the system-level events occurring in a software real-time kernel. Such events are produced by inserting software instructions (software probes) that writes to a PU register connected to the system/processor bus. Software probes can also be used as checkpoints in the code (*flags*), or to report memory contents.

Not only may system-level events be monitored, but also the hardware logic itself can be analysed and depicted against higher-level events, e.g. in waveform graph tools. This feature is useful for tracking down hardware logic errors which cannot be analysed using conventional probing methods (e.g. logic analysers and oscilloscope).

### 9.2.1 The Probe Unit

Figure 9.3 shows a block-diagram of PU's internal organisation. The component illustrated in top of the figure is the *event-detector* which merely performs conditional comparisons (comparator) on input signals. The input signals are hard-wired (in HDL) from selected points in the SoC. Also the condition expressions that defines events are hard-coded in the event-detector. When a certain condition for an event is detected, a sample is collected and stored immediately along with a timestamp in an on-chip memory buffer. Events generated by software, i.e. software probes, are detected as write-accesses to a 32-bit register (SWPROBE\_REG) in the PU's bus-interface (address/data/control).

An event-sample comprises the event-type, the timestamp, and an event-defined parameter field, see Figure 9.4. The parameter field is used to store additional information about an event. For instance, for various access-events to IP-components to be enough informative, the parameter field might contain call parameters, e.g. a bus-vector. The parameter field for a software-probe constitutes the 32-bit value that was written to SWPROBE\_REG.

The timestamp comes from a 32-bit timer device which denotes the rela-

tive system time with a resolution of  $1 \mu\text{s}$  per tick. A default timer device is included in the PU.

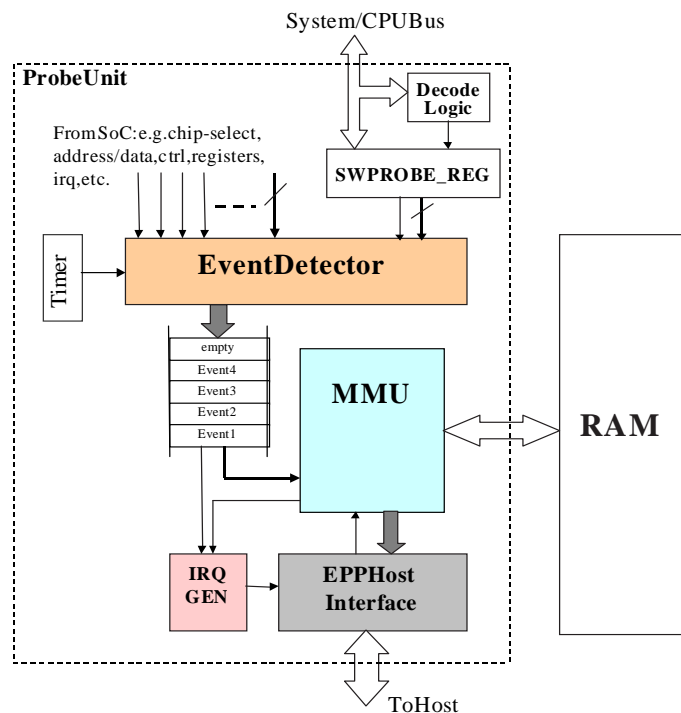


Figure 9.3: Internal organisation of the Probe Unit

Since the amount of event samples can be rather large before it can be communicated to a host computer, the on-chip memory requirements may not be feasible because of area or economical constraints, especially in FPGA SoCs. Therefore an external (off-chip) RAM buffer could be required. If an external RAM buffer is used, the on-chip buffer would still be needed in order to avoid write-access latencies to the external RAM. With this buffer configuration it is possible to detect and store up to 5 events occurring within  $1 \mu\text{s}$ , if a clock speed of 10 MHz is assumed. This is quite useful in some extreme situations, for instance, when monitoring timing-dependant response to external interrupts when there are several sources of competing interrupts.

The Memory Management Unit (MMU) is responsible for moving event samples from the internal buffer to the external RAM. Both buffers are organised as circular FIFO buffers for maximum space utilisation. The current version of MAMon is implemented with a 128kB RAM buffer. With an event sample size of 10 bytes (Figure 9.4) this means that more than 13 000 events can be stored before the buffer needs to be emptied. Furthermore, the MMU manages requests to move sample data from RAM to the host computer by way of the parallel port interface (described in next section).

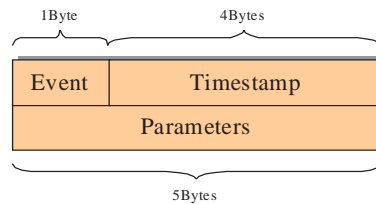


Figure 9.4: The event sample format

### 9.2.2 Host interface

Since the on-board event buffer is limited it is important that event samples are transferred to the host with a guaranteed high communication bandwidth. Therefore, a parallel port implementing the bi-directional *Enhanced Parallel Port* protocol (EPP 1.9 [EPP]) is used as the host communication interface. With EPP the event samples can be transferred with a rate up to 2MB/s, or more than 200k events per second.

As part of the host interface is the *interrupt generator*, refer to Figure 9.3. This component can be programmed to interrupt the host computer whenever there are new events in the buffer. When enabled, the interrupt generator can be set into one of three modes:

- Interrupt whenever new events are detected
- Interrupt when the RAM buffer is half-full
- Interrupt when the RAM buffer is full

The first two modes are useful when continuous monitoring is desired. The third mode is more useful if the PU is set to sample from a given command

until the buffer becomes full, and then stop. Providing the ability to choose the interrupt mode gives a customised solution that best suits the capabilities of the host computer performance, the tools, or the user. When the interrupt function is disabled, events can still be acquired in *polled* mode. Control of the PU's behaviour, and acquisition of event samples and other status information, is all done via the EPP register interface, i.e. from the host computer system.

### 9.2.3 The tool environment

The proposed tool environment provides the user with facilities to view and search the event samples received from PU. In order to manage possibly huge amounts of events that can be produced from long system runs, the received data must be stored in file-structures that are optimal for searching. A database will therefore be used for storage of the event samples. As illustrated in Figure 9.5, the database then acts as a server for various requests from the tools.

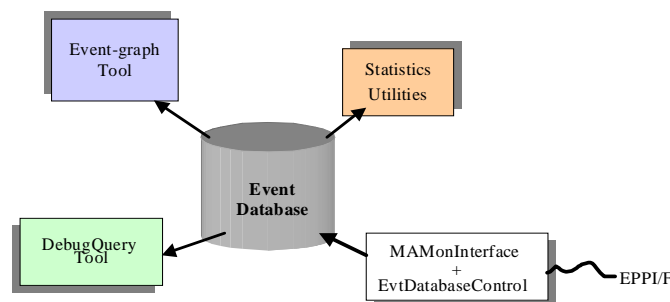


Figure 9.5: MAMon's tool environment

An event-graph that displays portions of the event history is a necessity in order to help the user in finding erroneous execution patterns. The event-graph tool, illustrated in Figure 9.6, collects events from the database, and displays them along a timeline. Apart from standard functions such as zooming and scrolling, there is also support for time-markers that are used for timing measurements, and search-markers that can be used to locate event conditions and patterns. For logic-level events, the tool looks and behaves similar to a waveform graph. The difference however, is that the tool is able to show a mix between logic- and system-level information on the same timeline, giving the

user the ability to correlate events in the hardware and software.

In order to ease visibility, and understandability of the execution, an event-filter can be used to hide excess information. The filter-tool can also reduce the search-space which will improve performance of the database.

The event-database is also suitable for other post-analysis, such as extraction of various statistics. Examples on such applications are; diagrams and histograms showing task's execution-time, processor utilisation, IPC frequencies, interrupt-response times, etc.

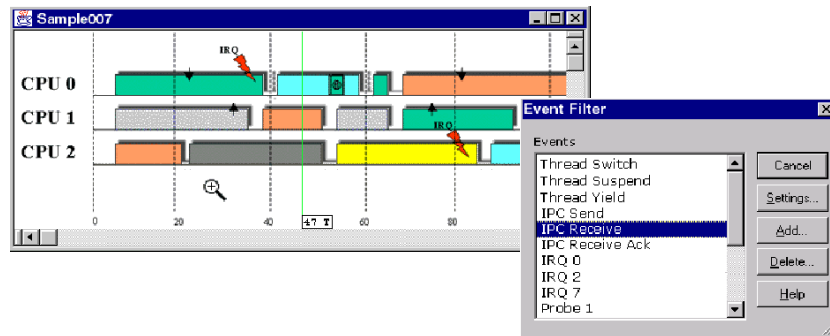


Figure 9.6: Example tools: Event-graph and event-filter

### 9.3 *An Ideal Example: Monitoring a Hardware Real-Time Kernel*

When connecting the MAMon system to a hardware-based RTOS kernel, the process-level can be extracted with zero software overhead, and thus, without changing the timing behaviour of the system. A hardware RTOS kernel implements traditional (software) RTOS functions (e.g. scheduling algorithms, task management, inter-process communication primitives, synchronisation, semaphores, event flags, etc.) in hardware. The RTU (Real-Time Unit [AFLS96, FSLA98]) is such a component that has proven to be successful for increasing RTOS performance and operational predictability. Moreover, it can also be used as the single RTOS for both single- and multiprocessor systems [LKF99].

The connection between MAMon's Probe Unit and the RTU is done using



signals in VHDL ([SL97]). The recorded events are then transferred to the host-based tool environment, where a process-view can be displayed using the event-graph tool. The provided facilities in the event-graph tool can help the designer to find erroneous execution patterns and/or be used to tune performance and load-balance in a multiprocessor SoC.

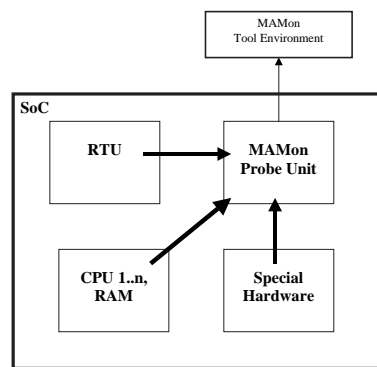


Figure 9.7: A RTL and system-level monitoring configuration using MAMon

Figure 9.7 illustrates a SoC configuration using the RTU for scheduling of one or more CPUs, and MAMon for monitoring the system at both the RTL and system-level. In this configuration no additional monitoring software is required on the target.

#### 9.4 Current and Further Work

A first version of MAMon is currently being implemented together with a simple tool environment including the event-graph display tool. In this version only tasking activities inside the RTU are monitored. Further work will primarily focus on adding support for monitoring the other RTU features, such as inter-process communication management, and handling external interrupts. Moreover, there will be extensive work on development and improvement of the tool environment. The results will be published in a forthcoming paper, together with a case study showing the use of MAMon for debugging of typical timing and synchronisation errors. There are also plans to extend the Probe Unit with support for run-time detection of user-defined event-patterns which then can be used to halt the software, either completely (all CPUs) or partially

as per CPU or task, or groups of CPUs and tasks. This feature could be utilised to implement synchronous and consistently halting breakpoints for use in co-operation with traditional source-level debuggers.

## 9.5 Conclusions

On-chip support for monitoring and debugging is becoming critically important since traditional solutions that uses in-circuit emulation (ICE) techniques, logic analysers, and oscilloscope, do not keep pace with today's system speeds. Moreover, on-chip approaches are motivated because of limited pinouts in chip-packaging, and even difficulties in reaching the physical pins (e.g. Ball-Grid Arrays, BGA).

The approach of integrating MAMon on a SoC offer an on-chip solution that also gives non-intrusive, synchronous, and consistent RTL and system-level monitoring. This, in turn, is ideal for event-based debugging and profiling of embedded real-time SoC applications. MAMon together with a hardware RTOS kernel gives a simple solution to process-level monitoring without requiring additional software overhead.

In a FPGA solution it is also convenient to monitor a mix between RTL system-level to get an effective debug and optimisation environment. In an ASIC solution MAMon is more suitable for run-time system-level monitoring, e.g. for process views.

All monitored data are time-stamped with a resolution of 10 times the system clock frequency. This capability is important so that events at different abstraction-levels can be compared and correlated with a high precision.

The requirement to manually connect SoC logic and signals to MAMon's Probe Unit, and then define event-conditions, is inconvenient and can be tricky to handle for SoCs with many small submodules. Here, it is desirable to have a tool, preferable an interactive GUI, that automates the necessary connections and definitions in HDL. In this case, the HDL code would probably need pre-processing before compilation and synthesis.

## Bibliography

- [AFLS96] Joakim Adomat, Johan Furunäs, Lennart Lindh, and Johan Stärner. Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, June 1996.

- [Alt] Altera corporation. <http://www.altera.com/>. 101 Innovation Drive, San Jose, CA 95134.
- [Baw99] H. Bawtree. Real-time monitoring with StethoScope 5.1. *Software Development*, 7(9), September 1999.
- [EPP] Enhanced parallel port v. 1.9. IEEE 1284.
- [FSLA98] Johan Furunäs, Johan Stärner, Lennart Lindh, and Joakim Adomat. *RTU94 - Real-Time Unit 1994 - Reference Manual*. Computer Architecture Lab, Dept. of Computer Engineering, Mälardalen University, Västerås, Sweden, January 1998.
- [Gai86] Jason Gait. A probe effect in concurrent programs. *Software - Practise and Experience*, 16(3):225–233, March 1986.
- [LKF99] Lennart Lindh, Tommy Klewin, and Johan Furunäs. Scalable architectures for real-time applications - SARA. In *CAD & CG'99*, December 1999.
- [Men98] Mentor Graphics, Microtec Division. *XPERT Profiler - Measurement and Evaluation Tool*, 1998.
- [MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–621, December 1989.
- [SL97] Stefan Sjöholm and Lennart Lindh. *VHDL for Designers (473 pages)*. Prentice-Hall, January 1997.
- [TBHS96] Jeffrey J.P. Tsai, Yaodong Bi, Steve J.H. Hang, and Ross A.W. Smith. *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis*. Wiley-Interscience, 1996.
- [Xil] Xilinx inc. [http://www.xilinx.com/prs\\_rls/ibmpartner.htm](http://www.xilinx.com/prs_rls/ibmpartner.htm). 2100 Logic Drive, San Jose, CA 95124-3400.



---

## Chapter 10

# **Paper B: On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems**

---

Mohammed El Shobaki

In proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA), Tokyo, Japan, March 2002.

## Abstract

This paper presents a novel hardware monitoring system that gives non-intrusive observability into the execution of hardware-accelerated Real-Time Operating Systems.

Monitoring is a necessity for testing, debugging and performance evaluations of real-time computer systems. Most research into monitoring of real-time systems have been devoted to minimising the execution interference imposed by the monitor. One approach to this has been the use of hardware support to extract software execution traces by probing the external processor (or system) busses.

However, the use of cache memories on various levels, and the increased integration of system components on-chip (SoCs) in addition to limited chip-package pins, severely obstructs traditional hardware monitors from probing processor signals and busses. For real-time systems built on these premises there is a need to access execution information residing on-chip, as well as to avoid interference with the system's execution behaviour.

In this paper we present an integrated solution to on-chip monitoring of system-level events in a real-time system. The monitor, called MAMon<sup>1</sup>, probes a hardware-based Real-Time Kernel using a Probe Unit integrated as an IP-block. This component detects and collects events regarding process' execution, communication, synchronisation, and I/O interrupt activities. Collected events are timestamped and transferred to a separate computer system hosting an event database and a set of monitoring application tools. We describe the monitor architecture, the implementation of a prototype, and an evaluation of its use.

---

<sup>1</sup>Multiprocess Application Monitor

## 10.1 Introduction

Run-time observability in embedded system architectures is a requirement for testing, debugging, and for validating design assumptions made about the behaviour of the system and its environment. The classical approach to run-time observability is to apply *monitoring*, i.e. the process of detecting, collecting, and interpreting run-time information regarding the system's execution behaviour. In monitoring real-time systems an important aspect is to minimise, or completely avoid, the intrusiveness of the monitor on the system's timing and execution properties. Failing to handle monitor intrusivity may lead to *probe effects* which cause non-deterministic behaviour in programs with race conditions and poor synchronisation [Gai86, MH89].

The research efforts on real-time monitoring has over the past decade been mostly devoted to dealing with probe effects and timing interference in various applications of monitoring [TKM89, TFC90, CJD91, JRR94, HW90]. Hence, a wide spectrum of monitoring approaches have been proposed, ranging from pure software techniques [TKM89, JRR94] to the use of special hardware support [LP89, TFC90, HW90]. *Software monitoring systems* offer the cheapest and most flexible solution where a common technique is to insert instrumentation code at interesting points in the target software. When the instrumentation code is executed the monitoring process is triggered and information of interest is captured into trace buffers in target system memory. The drawbacks of instrumentation is the utilisation of target resources such as memory space and processor execution time. Moreover, to avoid probe effects, the instrumentation code must be kept in the deployed software or be compensated for in the real-time schedulability analysis [TKM89] - with both alternatives resulting in performance penalties. *Hardware monitoring systems* on the other hand use special hardware to passively probe the target's physical busses, such as the processor and system busses, and collect information of interest without interfering with the target's execution. The main advantage with hardware monitoring is that probe effects can be completely avoided. The disadvantages are the dependancy on the target architecture and its related costs. *Hybrid monitoring* uses a combination of software and hardware monitoring and is typically used to reduce the impact of software instrumentation alone [HW90].

With today's highly integrated hardware, encapsulating complete systems on a chip (SoC), the traditional hardware monitors are facing severe difficulties. Processor cores, I/O components, cache memories, and even standard memory, are all integrated on the same chip. Given also that chip packages can be obstructive (as in Ball-Grid Array packages) and have limited pins, it has

become almost impossible for external hardware to probe internal signals. For real-time systems built on these premises there is a need to access execution information residing on-chip, as well as to avoid interference with the system's execution behaviour.

In this paper we present an architecture for on-chip monitoring of single- and multiprocessor real-time systems that are based on hardware-accelerated operating systems [AFLS96, LKF99, MRS<sup>+</sup>90, NUI<sup>+</sup>95]. The monitor, called MAMon, probes a hardware-implemented Real-Time Kernel (RTK) using a Probe Unit integrated as an IP-block at the VHDL-level. A hardware RTK implements traditional (software) RTOS functions, such as scheduling algorithms, process management and communication, in hardware [AFLS96, LSF<sup>+</sup>98]. Operating at the system-level the Integrated Probe Unit detects and collects events regarding process' execution, communication, synchronisation, and I/O interrupt activities. The collected events are timestamped with the resolution of the system clock frequency (10 MHz = 100ns) and then transferred, via a high-speed parallel port link, to a separate host computer system. At the host the events are stored in a database which constitutes the heart of a monitoring application framework featuring event analysis and debugging (searching, filtering, and graphing), performance evaluations, and more. Monitoring occurs mainly at the system-level, but lower abstraction-levels are supported too by allowing instrumentation code to write to dedicated *probe registers* in the monitor hardware. This opportunity would, however, classify the monitor as a hybrid system, and thus requires a perturbation analysis of the software instrumentation.

The main contributions of this work are the ideas on system-level monitoring of hardware RTKs, on-chip rather than by probing external processor busses. We believe that on-chip monitoring support will be required in future development of real-time systems, especially those based on SoCs.

The paper is organised as follows. Section 10.2 describes a multiprocessor system concept based on a hardware-accelerated RTOS. This system will be the target platform in further discussions on our proposed monitor. Section 10.3 describes the monitor architecture for a generic target RTOS that utilise hardware RTKs. An overview of the system and a detailed description is given for the Integrated Probe Unit, the host-based monitoring application framework, and the communication interface in between. Section 10.4 describes an FPGA prototype implementation of the monitor for a multiprocessor system with 3 PowerPC-750 processors. An evaluation of the prototype is given in Section 10.5, and finally, Section 10.6 summarises the paper with some con-



cluding remarks and directions on future work.

### 10.2 A Real-Time Multiprocessor Architecture - SARA

The Scalable Architecture for Real-Time Applications (SARA [LKF99, KL99]) is a research platform for real-time multiprocessor computing systems. The two main research objectives with SARA are: 1) to provide a hardware architecture that behaves predictably to the real-time application, and 2) to provide a flexible system architecture that simplifies processor (performance) scalability. In attaining these design goals, a SARA architecture is based on a *hardware-accelerated* RTOS. The hardware support comes from a co-processor called RTU (Real-Time Unit [AFLS96, LSF+98]) which provides the RTOS with kernel-level services such as process/task scheduling, synchronisation and communication, see Section 10.2.1 for more details.

Figure 10.1 shows the hardware view of a SARA system which includes one or more processor nodes, a communication network (bus), and the RTU as a shared software process scheduler. This view is the same whether the hardware is implemented on a multi-board computer system, such as VME or CompactPCI-based [PCI] systems, or as a SoC. A SARA implementation on a CompactPCI system is described in Section 10.2.2, and in [CHNA01] a SoC implementation is proposed.

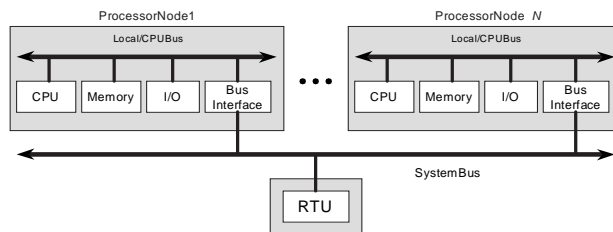


Figure 10.1: Hardware view of a SARA system

The software, which is partitioned onto each node in SARA, includes a *minimal* RTOS which mainly interfaces to the RTU, and a collection of processes which are scheduled to execute on one or more processor node(s). To simplify the programming model, hardware is abstracted to the software so that processes need not be bound to a certain processor, and process migration is allowed.

Communication between processes takes part over a virtual bus (VCB) which spans over all processor nodes. The VCB programming model, shown in Figure 10.2, uses the concept of *virtual slots* which processes must attach to in order to send and receive messages. Moreover, synchronised sending, broadcasting and multicasting of messages is supported.

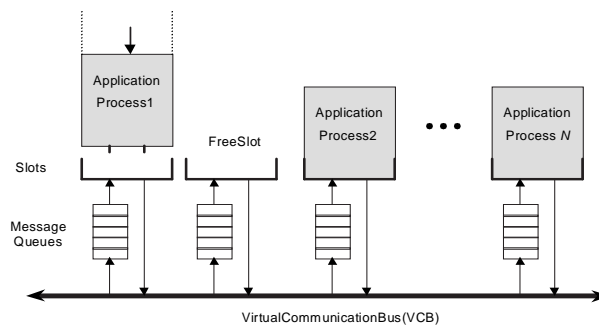


Figure 10.2: Process communication model in SARA

### 10.2.1 RTU - Real-Time Kernel in Hardware

Hardware support to increase performance and predictability in real-time operating systems have been proposed in [MRS<sup>+</sup>90, AFLS96, LSF<sup>+</sup>98, NUI<sup>+</sup>95]. The Real-Time Unit, RTU by Lindh et. al. [AFLS96, LSF<sup>+</sup>98], is a co-processor with support for real-time kernel services such as process scheduling and management (create, terminate, etc), inter-process communication (IPC, message send/receive), synchronisation (semaphores), and I/O interrupt handling. The RTU, which supports scheduling of both single- and multiprocessor systems, runs in parallel with the target system's processor(s). Processors interface with the RTU by memory-mapping to its processor-independent register interface. Via this interface, *service-calls* are placed by writing to dedicated service-call registers.

Figure 10.3 shows the basic building blocks of the RTU. The core part is the scheduler which schedules processes on-line (pre-emptive priority scheme) and dispatches process execution. Connected in between the scheduler and the programming/bus interface, a set of functional modules implements the various services in the RTU, such as management of the scheduler, IPC, semaphores,

clock and timer management. Process context-switching is notified to CPUs using interrupts causing handlers in software to perform the actual context-switching.

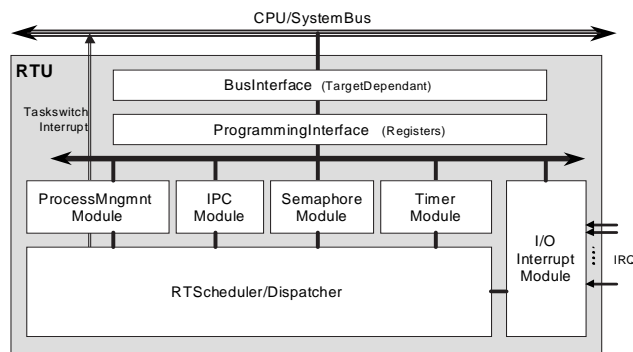


Figure 10.3: Basic building blocks of the RTU

### 10.2.2 A SARA CompactPCI System

Described in [KL99] is a SARA implementation on a CompactPCI (CPCI [PCI]) computer system. A CPCI system has 8 slots where CPU-boards can be inserted. The first slot, slot 0, is dedicated as the *system slot* which requires that the CPU-board on that slot handles arbitration and clock distribution on the CPCI backplane. Figure 10.4 shows the current SARA implementation with 3 PowerPC-750 CPU-boards. The RTU, which resides on a PMC-board (PCI Mezzanine Card [PCI]), is attached to the system board from where it can communicate with all CPUs in the system (see also Figure 10.10).

All CPU-boards have local memory and a local PCI-bus. Processes that are allowed to migrate between CPUs require global memory to hold their Process Control Blocks (PCB). This global memory can be defined out of local memories on all CPU-boards. Currently, global memory is allocated at the system board only.

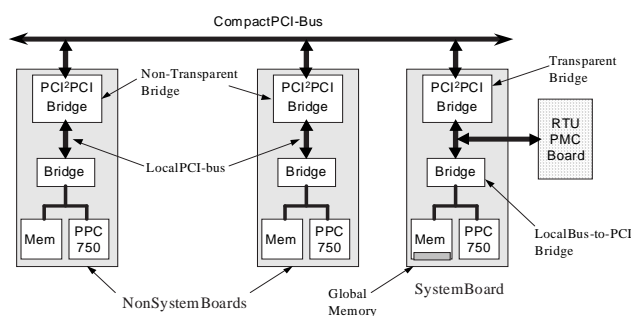


Figure 10.4: A SARA system based on CompactPCI-board computers [KL99]

### 10.3 A Monitoring System for Hardware-Accelerated RTOSs

#### 10.3.1 Overview

The proposed monitoring system aims at providing means for on-chip observability at the system-level in single- and multiprocessor real-time systems. The monitor, which we call MAMon (short for Multiprocess Application Monitor), is based on the following MAMon assumptions about the monitored target system:

- The target's RTOS is supported by a hardware Real-Time Kernel (RTK), like the RTU or a similar component as described in Section 10.2.1.
- The RTK holds information about the state of every process in the system, inter-process communication activities, timers, interrupts, etc.
- The RTK must allow external access to internal (vital) signals and data. Since the RTU was available to us as a *soft* IP-component (HDL source), access to all signals and data is straightforward in VHDL.

The architecture of MAMon, shown in Figure 10.5, consists of two major parts: the *Integrated Probe Unit* (IPU, Section 10.3.2) which is the hardware part of MAMon, and a *host* computer system. Like an IP-block, the IPU is integrated with the hardware RTK at the VHDL level. In a SoC the IPU may also be connected to processor busses, I/O components, and other hardware logic in order to extract information at various levels of abstraction. In the

synthesized hardware (e.g. ASIC or FPGA implementation), the IPU monitors the RTK in run-time, and collects events regarding the system-level behaviour of the real-time application. The collected events are timestamped each and then transferred over a high-speed parallel communication port to the host computer where they are stored in a database. In an integrated framework (Section 10.3.5) the database serves as an event repository which can be used by monitoring application tools to provide event-based debugging, performance analysis, assessment of design constraints, etc.

In certain cases there is a need to generate events from software, for instance, to mark code checkpoints (flags), or to report register and memory contents required for lower-level analysis. Such events are produced by inserting software instructions (software probes) that writes to a dedicated register connected in between the IPU and the system/processor bus.

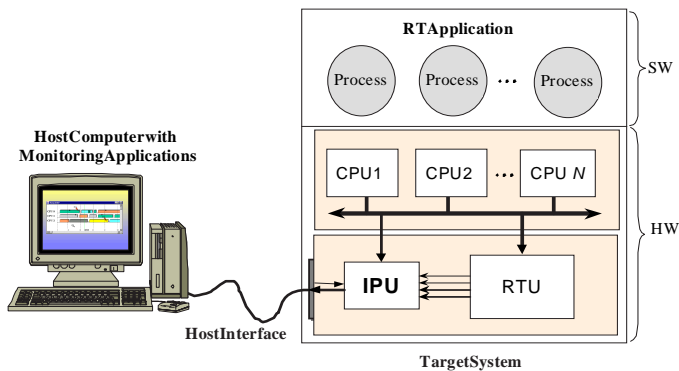


Figure 10.5: Overview of MAMon

### 10.3.2 The Integrated Probe Unit

Figure 10.6 shows a block-diagram of the IPU's internal organisation.

#### 10.3.2.1 Event Detector

The *Event Detector*, seen in top of Figure 10.6, is responsible for the detection of events and for collecting event samples. Detection of events is performed by comparing input event signals with pre-defined event *condition expressions*. The input signals are hard-wired (in HDL) from selected points in

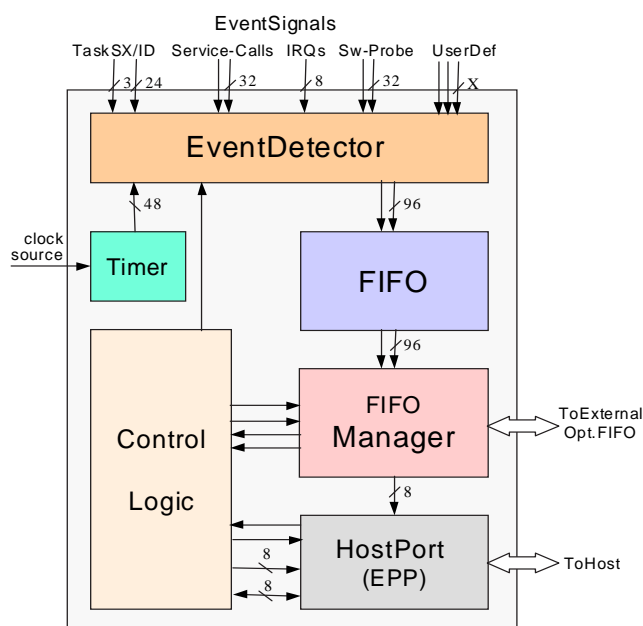


Figure 10.6: The Integrated Probe Unit

the RTU. Over-sampling of the input signals is not required because the pre-defined events will never occur simultaneously. When an event is detected, a sample is collected and stored immediately along with a timestamp in the local FIFO buffer.

An event-sample comprises the event-type, the timestamp, and an event-defined parameter field, see Figure 10.7. The parameter field is used to store additional information about an event. For instance, for a *task-switch event* to be sufficiently informative, the parameter field contains the new task's id-number and the CPU it was scheduled to run on. For a *send message event*, the parameter field may contain the id-number of the receiving task and the pointer to the message, and so on.

The timestamp comes from the 48-bit Timer module which denotes the absolute system time given in nanoseconds. The Timer is updated at the resolution of the system clock frequency.

To support detection of software probes, the IPU provides a simple inter-

face that can be used by external decode logic. A single strobe line is all that is required to indicate a software write-access, and to signal the IPU to latch incoming data.

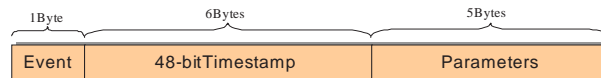


Figure 10.7: The event sample format

#### 10.3.2.2 FIFO

The FIFO buffer is needed during transient over-loads of events while the host computer is busy reading event data over the parallel port. FIFO buffer dimensioning is described in Section 10.3.4. The FIFO is built onto on-chip dual-ported RAM with parameterizable (generic) size and port width. Moreover, it has signals that indicate when the buffer becomes full, or half-full.

#### 10.3.2.3 FIFO Manager

The FIFO Manager mainly provides a byte-wide interface for the Host Port to read event data from the FIFO. In circumstances when the required FIFO size is not feasible on-chip, e.g. in FPGA implementations, the FIFO Manager can also be used to extend the FIFO using external RAM. In this case, the FIFO Manager will also take care of flushing the contents of the on-chip FIFO out to the external RAM. The option to use external RAM can be set via the Host Port.

#### 10.3.2.4 Host Port

The Host Port is responsible for taking care of host-initiated acquisition of event data. It also provides the host with a programming interface to read the status of the IPU and to control its behaviour (the Control Logic in Figure 10.6).

Since FIFO buffering is limited it is important that event samples are transferred to the host with a guaranteed high communication bandwidth. Therefore, the Host Port implements the bi-directional *Enhanced Parallel Port* protocol (EPP 1.9 [EPP]). In theory the EPP supports transfer rates up to 2MB/s (approx. 160k events/s).

To indicate availability of events in the FIFO the Host Port can be programmed to generate an interrupt to the host computer. When this feature is

enabled, it can be set into one of three modes:

- Interrupt whenever new events arrive
- Interrupt when the FIFO buffer is half-full
- Interrupt when the FIFO buffer is full

The first two modes are useful when continuous monitoring is desired. The third mode is more useful if the IPU is set to sample from a given command until the FIFO becomes full, and then stop. Providing the ability to choose the interrupt mode gives a customised solution that best suits the capabilities of the host computer performance, the tools, or the user. When the interrupt function is disabled, events can still be acquired in *polled* mode.

### 10.3.3 Events

Currently the Event Detector supports detection of four types of events; *Taskswitches*, *Service-Calls*, *Interrupts*, and *Software Probes*. The conditions for these events are hardcoded in the Event Detector. Therefore, the size of the Event Detector logic is linearly proportional to the number of supported events. Given below is a description of each event-type; its condition(s) and related data to be collected.

#### 10.3.3.1 Taskswitch events

For a taskswitch to be detected, the IPU is connected directly to the scheduler module in the RTU. Whenever a taskswitch is to occur, the scheduler asserts an interrupt signal and indicates the next task's id along with the CPU it is to run on. Upon detection of this event the following packet is produced.

<i>TSW_EVT</i>	<i>TIMESTAMP</i>	<i>CPU_NR</i>	–	<i>TASK_ID</i>
1B	6B	1B	2B	2B

#### 10.3.3.2 Service-Call events

A service-call is detected whenever software writes to a *Service-Call Register* in the RTU, i.e. to indicate a service-request. For each CPU in the system there exists one Service-Call Register in the RTU's register-interface. These registers are connected to the IPU as well. An event of this type produces the following packet.



<i>SVC_EVT</i>	<i>TIMESTAMP</i>	<i>CPU_NR</i>	<i>REG_VALUE</i>
1B	6B	1B	4B

**10.3.3.3 Interrupt events**

The RTU supports handling of external interrupts by associating tasks with the interrupts. When an interrupt is asserted the RTU’s interrupt module tells the scheduler to start the associated task. To detect this event, the interrupt lines are connected to the IPU along with the associated tasks’ id. An interrupt event produce the following packet.

<i>IRQ_EVT</i>	<i>TIMESTAMP</i>	<i>IRQ_NR</i>	-	<i>TASK_ID</i>
1B	6B	1B	2B	2B

**10.3.3.4 Software-Probe events**

A software probe is similar to a service-call request in that software writes to registers in the RTU. However, these register are dedicated to MAMon and are connected only to the IPU. Values written to these registers can be used for profiling, measurements and debugging purposes. A software probe event produce the following packet.

<i>SWP_EVT</i>	<i>TIMESTAMP</i>	<i>REG_NR</i>	<i>REG_VALUE</i>
1B	6B	1B	4B

**10.3.4 Performance and FIFO Dimensioning**

**10.3.4.1 Input rate**

The rate at which the EDU detects and stores events in the FIFO buffer depends on the system frequency; the higher frequency, the higher the input rate to the buffer. The EDU requires 2 clock cycles to store one event in the buffer. Since the currently supported events (see previous section) cannot occur consecutively within 2 clock cycles, no events will be missed. This implies that the worst condition corresponds to an event occuring every 2 clock cycles. With a clock frequency of 10 MHz, the input rate is 1 occurrence per 200 ns. It is also assumed that the input rate follows a Poisson statistical distribution, as described in [LP89].

**10.3.4.2 Output rate**

The output rate for emptying the event FIFO buffer is largely determined by the performance of the host interface communication link, the EPP port in this

case. In theory, EPP supports transfer rates up to 2 MB/s [EPP]. When using a PC running Linux as the MAMon host computer, and a standard bi-directional parallel port interface, we could reach a maximum transfer rate of 1.3 MB/s. We therefore estimate the time to transfer one event-packet to 10  $\mu$ s, assuming a transfer involves 13 byte reads; 12 for event data plus 1 for reading the IPU's status register (to check for event availability). The rate at which events are stored in the database is not considered since it is much faster than the communication bandwidth.

#### 10.3.4.3 FIFO buffer dimension

To eliminate buffer overflow the FIFO buffer must be large enough to handle the worst case input flow while the MAMon host system is busy flushing the buffer. By applying a queueing analysis (adopted from [LP89]) we can estimate the required buffer size. This analysis assumes two facts: 1) events can arrive concurrently while the buffer is flushed, and 2) that the MAMon host system starts flushing the buffer at latest when the buffer is *half* full. The first assumption is fulfilled as the FIFO buffer is dual-ported. The second assumption requires that the host either polls continuously for new events, or uses the half-full-buffer interrupt mode.

Let  $k$  be half the FIFO buffer size,  $R$  the mean input rate, and  $T$  the transfer time per event-packet. Assuming that the input rate follows the Poisson distribution, then,  $\mathcal{P}(k)$  is defined as the probability that the buffer has  $k$  arrivals in time  $T$  (i.e. that half the buffer fills up within  $T$ ). The probability function is,

$$\mathcal{P}(k) = \frac{(RT)^k}{k!} e^{-RT}$$

In determining the total buffer size ( $2k$ ) it is assumed that the probability of filling up half the buffer is at a minimum, for instance 0.5%, given that it takes  $kT$  time to flush the first buffer half. That is,

$$\mathcal{P}(k) + \mathcal{P}(k+1) + \mathcal{P}(k+2) + \dots < 0.005$$

Using  $T = 10\mu$ s and  $R = 1/200$ ns, gives 70 as the best value for  $k$ . Hence, the FIFO buffer must handle no less than 140 events.

#### 10.3.5 The Monitoring Application Framework

To provide the user with a platform for event-based performance analysis and debugging, we have developed an integrated framework for monitoring appli-

cations. Our goal is not to develop a complete monitoring environment, but to show the capabilities with our hardware monitoring approach.

The framework is developed mainly in Java and uses an SQL database to store the event histories. Figure 10.8 shows this framework's architecture. At the bottom lies the IPU interface module which is mainly used to transfer event samples from the IPU into the SQL database, and to control the behaviour of the IPU. The IPU interface module runs as a separate process, but is controlled from the Java framework via the Java Native method Interface (JNI). JNI is required because this module is written in C/C++ as it is strongly dependant on the underlying architecture for communicating with the EPP interface. The SQL database is run by the MySQL DBMS ([www.mysql.com](http://www.mysql.com)). We choose MySQL for its speed and capabilities to handle our amounts of events, and because it is free for educational purposes. The database and IPU interface constitutes the base of our framework.

The Java application forms the actual framework which provides an integrated interface to control the monitoring process, to collect events into the database, and to query the event database in various ways. Using this interface we can now easily implement application specific monitoring tools that are plugged into the framework.

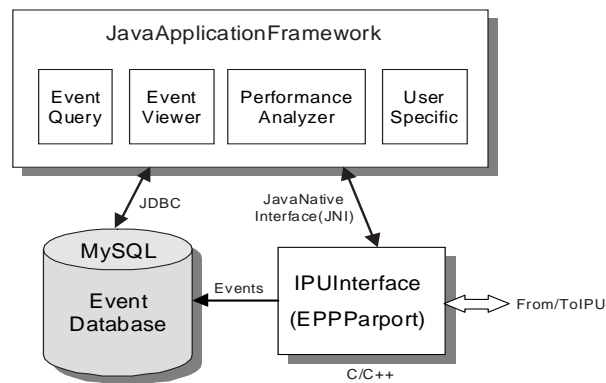


Figure 10.8: MAMon's Application Framework

An example monitoring tool is the *Event Viewer* that displays portions of the event history. Such a tool can be useful for finding and analysing erroneous execution patterns. The Event Viewer tool, shown in Figure 10.9, col-

lects events from the database, and displays them along a timeline. Apart from standard functions such as zooming and scrolling, there is also support for time-markers that are used for timing measurements, and search-markers that can be used to locate event conditions and patterns.

Another example tool is the *Event Query* tool (also shown in Figure 10.9) which provides a user-friendly interface to query the database for event conditions and execution patterns. The output from the query may be output textually to screen or to a file, or graphically by linking its results with the Event Viewer tool.

The event database is also suitable for other post-analysis, such as extraction of performance indexes for use in diagrams and histograms showing task's execution time, processor utilisation, IPC frequencies, interrupt response times, etc.

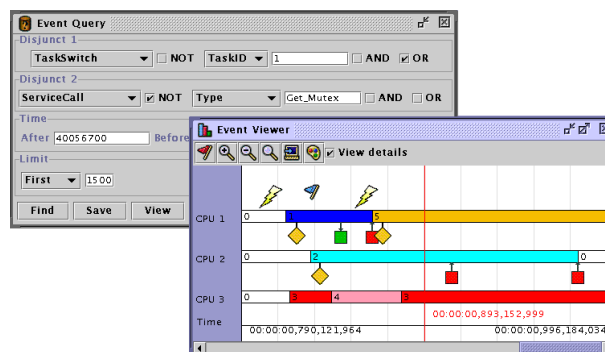


Figure 10.9: Screenshot of Event Query & Event Viewer tools

## 10.4 Physical Hardware Implementation

In this section we present some implementation details on a prototype of MAMon for a SARA CompactPCI system (described in Section 10.2.2).

### 10.4.1 The Hardware Prototype

The IPU is implemented together with the Real-Time Unit<sup>2</sup> on a Xilinx Virtex-1000 FPGA [Xil]. All modules are designed in VHDL which is either textually entered or automatically generated from state and block diagrams drawn in Renoir (graphical hardware design tool, by Mentor Graphics). The FPGA is mounted on a PMC-board and connects to the SARA-system via a PCI-bus-interface chip (PLX-bridge), see Figure 10.10. The host system of MAMon connects to the parallel-port connector (left in Figure 10.10) with a IEEE-1284C cable [EPP]. Because RAM-cells are limited inside the FPGA, a 128kB SRAM module (on backside of the board) is used to extend the internal event FIFO buffer.

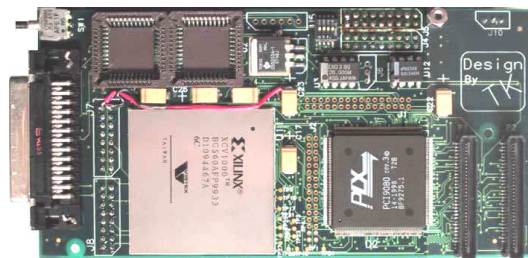


Figure 10.10: PMC-board with a Xilinx Virtex-1000 FPGA and PLX-bridge [KL99]

### 10.4.2 Physical Footprint

Table 10.1 shows some area figures from a synthesis to a Xilinx Virtex-1000 target. Although these figures are target-specific, they could serve as a reference for estimating the equivalent area requirements for other silicon technologies. Xilinx's FPGA technology can be described as matrices of Configurable Logic Blocks (CLBs) where each CLB contains two Function Generators (FGs) and two D-Flip-Flops. According to Xilinx, a Virtex-1000 FPGA has a capacity of "1 million gates" [Xil].

As shown in the table, the IPU makes up only 5% of the total number of CLBs for the whole design. What is not shown in the figures, however, is the area costs for the event FIFO buffer. This is because FIFO memory was mapped

<sup>2</sup>The RTU in this prototype was synthesised to support 128 tasks with 64 priorities.

<i>Resource</i>	<i>IPU</i>	<i>RTU</i>	<i>Avail</i>	<i>Utilisation</i>
CLB Slices	181	3276	12288	28.13%
FGs.	361	6553	24576	28.13%
Flip-Flops	254	2580	24576	11.53%

Table 10.1: Area figures for a Xilinx V1000 FPGA

onto RAM cells built-in the FPGA (called Block-Select RAM). However, calculating the area costs for memory is straightforward in many technologies. Currently the event FIFO buffer can store 16 events where each is 96 bits wide (12 bytes), i.e. 1536 bits are required.

The host interface port, currently implemented as an EPP parallel port, requires 15 I/O pins; 8 for data, and 7 for control. On a chip with limited pinouts it could be preferable to multiplex these pins with other I/O, or choose an interface with less ports, e.g. a synchronous serial port. Moreover, an additional 29 I/O pins are used to interface with the external SRAM used to extend the event FIFO buffer. As this memory is optional, this overhead can be removed if the internal event FIFO can be fitted on-chip.

## 10.5 *Prototype Evaluation*

The prototype system was validated in a number of small tests on both single- and multiprocessor targets. With no intrusion on neither the execution or the timing behaviour of the target system the prototype was able to monitor task-switches, service-calls, and external interrupts. Monitoring of software probes (hybrid monitoring) was also accomplished but with a minimal intrusion equal to the delay of a 32-bit PCI-transfer per probe (@33MHz = 30ns). To illustrate a proof of concept we present hereunder an example where we analyse a deadlock situation using the monitor.

### *Example: Deadlock Detection*

The program in this example illustrate a typical situation where two tasks need to synchronise before proceeding to a next step, in this case opening a pair of fuel valves. The deadlock occurs due to an in-planted synchronisation error between the two tasks T1 and T2 which execute on processors CPU1 and CPU2 respectively. Figure 10.11 shows the pseudo-code for the tasks. The tasks

synchronise with mutual sending and receiving of messages over the VCB (described in Section 10.2). Task T1 uses the blocking *sendwait()* call to send a message and wait for the other party to send as well. For a proper synchronisation, task T2 should also call *sendwait()*, but due to a programming error the *receive()* call was used instead. This results in a deadlock since T1 cannot resume, and T2 will get blocked the second time it calls *receive()*.

<pre> Global VCB Slot_T1;  Task T1() {   Slot_T1 = Connect_to_VCB();   LOOP {     Compute_X;     ...     Slot_T1.sendwait(slot_T2);     Open_valve1();     ...     Close_valve1();   } } </pre>	<pre> Global VCB Slot_T2;  Task T2() {   Slot_T2 = Connect_to_VCB();   LOOP {     Compute_Y;     ...     Slot_T2.receive(); // Bug!     Open_valve2();     ...     Close_valve2();   } } </pre>
---	---

Figure 10.11: Deadlock example in pseudo-code

To locate the bug, we first monitor the target system and collect the system-level events into the host database. The Event Query tool is then used to perform a filtered search in the event database. Using predicate disjuncts and conjuncts in the query we can easily find the first and last occurrences of the tasks of interest. Figure 10.12 shows a text-dump from the query tool. Rows 1-3 shows that T1 starts and attempts to connect to the VCB. Rows 4-6 shows the similar sequence for T2. The *sendwait()* call in T1 is mapped to the VCB primitives *VCB\_Put* and *VCB\_Get* seen on rows 7-8. After that T1 gets blocked, the IDLE task starts on that processor (row 9). At row 10, T2 receives the message from T1 without blocking, and later when it attempts to receive again at row 11 it gets blocked too. The same sequence of events can also be depicted by the Event Viewer tool, see Figure 10.13. Horizontal bars indicate executing tasks, and the icons beneath indicate service-calls.

## 10.6 Conclusions

This paper has described a monitoring system and its implementation for non-intrusive monitoring of real-time systems. The monitoring system, called MAMon,

##.	Event	Timestamp	CPU	Subtype/Parameters
1.	TaskSwitch	00:00:00,815,083,600	CPU1	T1
2.	ServiceCall	00:00:00,821,241,000	CPU1	VCB_Alloc 0x140101
3.	ServiceCall	00:00:00,821,255,200	CPU1	VCB_Open 0x020101
4.	TaskSwitch	00:00:00,828,930,200	CPU2	T2
5.	ServiceCall	00:00:00,834,384,300	CPU2	VCB_Alloc 0x140202
6.	ServiceCall	00:00:00,834,395,000	CPU2	VCB_Open 0x020202
7.	ServiceCall	00:00:00,846,497,300	CPU1	VCB_Put 0x107102
8.	ServiceCall	00:00:00,864,018,800	CPU1	VCB_Get 0x10F501
9.	TaskSwitch	00:00:00,864,044,800	CPU1	IDLE
10.	ServiceCall	00:00:00,908,777,500	CPU2	VCB_Get 0x108502
11.	ServiceCall	00:00:00,979,800,300	CPU2	VCB_Get 0x108402
12.	TaskSwitch	00:00:00,979,826,300	CPU2	IDLE

Figure 10.12: Text-dump in the Event Query Tool

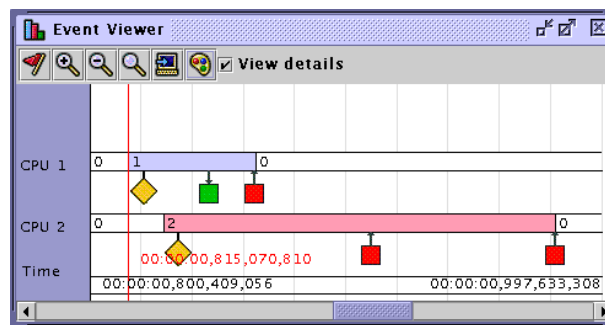


Figure 10.13: Deadlock seen in Event Viewer Tool

integrates a probe component with a hardware Real-Time Kernel in order to non-intrusively detect and collect process-level events at the target system. Via a parallel communication link, the collected events are transferred to a host computer system where they are stored in a database. Built onto the database, a set of monitoring applications provides post analysis features such as event-debugging, profiling, and performance evaluations based on the collected events. While our approach is non-intrusive, it also overcomes the difficulties in extracting execution information residing on-chip of processors and in Systems-on-Chip (SoC). Monitoring occurs at full system speeds, both in single- and multiprocessor targets. Although the monitor's probe component is hardware dependant in that it is tightly coupled to a hardware real-time kernel,



the solution is independent of target processor architectures and no software overhead is required.

The paper also describes the option to use MAMon as a hybrid monitoring system for monitoring at lower abstraction levels, e.g. functional and data levels. In this case instrumentation of the software is required and will introduce execution delays (although minimised).

To our knowledge, the proposed idea is novel and introduces a new alternative to monitoring, particularly useful in systems with hardware-accelerated real-time operating systems, and in SoCs. Future work includes further validation of the MAMon concept for monitoring of real-case applications, and for systems built on SoC hardware.

### *Bibliography*

- [AFLS96] Joakim Adomat, Johan Furunäs, Lennart Lindh, and Johan Stärner. Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, June 1996.
- [CHNA01] Mikael Collin, Raimo Haukilahti, Mladen Nikitovic, and Joakim Adomat. Socrates - a multiprocessor soc in 40 days. In *Proceedings of the Conference on Design, Automation and Test in Europe 2001, Designers Forum*, Munich, Germany, March 2001.
- [CJD91] Sarah E. Chodrow, Farnam Jahanian, and Marc Donner. Run-time monitoring of real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 74–83. IEEE, December 1991.
- [EPP] Enhanced parallel port v. 1.9. IEEE 1284.
- [Gai86] Jason Gait. A probe effect in concurrent programs. *Software - Practise and Experience*, 16(3):225–233, March 1986.
- [HW90] Dieter Haban and Dieter Wybraniec. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Trans. on Software Engineering*, 16(2):197–211, February 1990.
- [JRR94] Farnam Jahanian, Ragunathan Rajkumar, and Sitaram C.V. Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–273, November 1994.

- [KL99] Tommy Klewin and Lennart Lindh. Scalable architectures for real-time applications and use of bus-monitoring. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, China, December 1999.
- [LKF99] Lennart Lindh, Tommy Klewin, and Johan Furunäs. Scalable architectures for real-time applications - SARA. In *CAD & CG'99*, December 1999.
- [LP89] An-Chi Liu and Ranjani Parthasarathi. Hardware monitoring of a multiprocessor system. *IEEE Micro*, pages 44–51, October 1989.
- [LSF<sup>+</sup>98] Lennart Lindh, Johan Stärner, Johan Furunäs, Joakim Adomat, and Mohammed El Shobaki. Hardware accelerator for single and multiprocessor real-time operating systems. In *Proceedings of the Seventh Swedish Workshop on Computer Systems Architecture*, Göteborg, Sweden, June 1998.
- [MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–621, December 1989.
- [MRS<sup>+</sup>90] L. D. Molesky, Krithi Ramamritham, C. Shen, J. A. Stankovic, and G. Zlokapa. Implementing a predictable real-time multiprocessor kernel - the spring kernel. In *IEEE Workshop on Real-Time Operating Systems and Software*, May 1990.
- [NUI<sup>+</sup>95] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In *Proceedings of TRON'95*, pages 34–42, 1995.
- [PCI] PCI Industrial Computers Manufacturers Group. *CompactPCI Specification rev. 2.1*.
- [TFC90] Jeffrey J.P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen. A noninvasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, March 1990.
- [TKM89] H. Tokuda, M. Koreta, and C.W. Mercer. A real-time monitor for a distributed real-time operating system. *ACM Sigplan Notices*, 24(1):68–77, January 1989.

[Xil] Xilinx inc. [http://www.xilinx.com/prs\\_rls/ibmpartner.htm](http://www.xilinx.com/prs_rls/ibmpartner.htm). 2100  
Logic Drive, San Jose, CA 95124-3400.



---

## Chapter 11

# **Paper C: MAMon - A Multipurpose Application Monitor**

---

Mohammed El Shobaki and Jeroen Heijmans  
MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-121/2004-1-SE, Mälardalen  
Real-Time Research Centre, Mälardalen University, Västerås, Sweden, Sep-  
tember 2004.

### **Abstract**

This report describes the architecture and integral components of the Multipurpose Application Monitor (MAMon). The IPU's signal interface is described and we explain how the interface conforms to integration with a hardware RTK and the communication port to an external host computer. The report also provides a programmer guide for the monitoring application framework, as well as a user manual for the currently implemented monitoring tools within the framework. The report is mainly intended as a reference guide for working with MAMon at a user's level.

## 11.1 Introduction

This report describes the architecture and integral components of MAMon. The report also provide a programmer guide for the monitoring application framework, as well as a user manual for the currently implemented monitoring tools. The report is organised as follows:

**Section 11.2** Overview of MAMon

**Section 11.3** The Integrated Probe Unit

**Section 11.4** The MAMon Application Framework

**Section 11.5** Framework Software Architecture

**Section 11.6** Framework Programmer's Guide

**Section 11.7** MAMon Tool Desktop User Guide

### 11.1.1 Related documents

Some of the concepts discussed in this report are described in previously written documents. Thus, the following publications are recommended for additional information:

- Paper A: A Hardware and Software Monitor for High-Level System-on-Chip Verification [1]
- Paper B: On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems [2]
- HWMON – EPP/SPP IEEE1284 Host Interface [3]
- MAMON 1.1 USB – Implementation Report [4]
- Monitoring of System Work Load (M.Sc. thesis report) [5]

## 11.2 Overview of MAMon

The Multipurpose Application Monitor – MAMon for short – is divided in two major parts: 1) the *Integrated Probe Unit* (IPU) which is integrated with the observed computer system's hardware (the *target*), and 2) the *MAMon Application Framework* that constitutes all the software component residing on a separate computer system (the *host*). Figure 11.1 shows an overview of MAMon's

building blocks. In between the target and the host is the *Host Interface* which is a communication medium and a programming interface to the IPU. In principal, the IPU generates event packets that are sent to the host through the Host Interface, and vice versa, the host controls the IPU's behaviour through the same interface.

The MAMon Application Framework, or just *the Framework*, is a collection of software tools and components that together handles event storage, access, display and analysis. These include the *MAMon Tool Desktop* and the *monitoring tools* available in it, the SQL database where events are stored, and the *IPUBridge* which is a *driver* component that manages communication with the IPU. The MAMon Tool Desktop (MTD) may be seen as the central program that manages everything on the host, including the monitoring process at the target by way of the IPUBridge. Furthermore, the MTD controls the collection of event data into the SQL database, and hosts the tools that make use of the collected events. These tools reside in the MTD as *plug-ins* and provide various monitoring features, e.g. such as querying the event database for specific events or event patterns, display of event histories textually and graphically, save event histories to file, depict histograms and charts, etc. Moreover, the MTD and the plug-ins are programmed mainly using the Java object-oriented language.

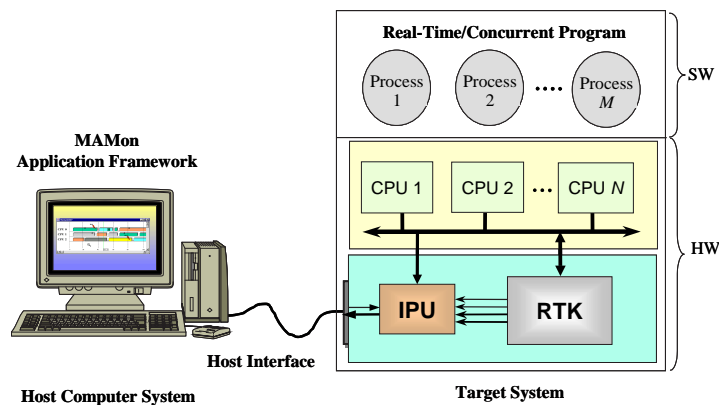


Figure 11.1: Overview of MAMon

The Framework and the MTD are overviewed further in Section 11.4, and in Section 11.5 the software architecture of the Framework is described. Sec-



tion 11.6 provide a manual for programming with the Framework. Section 11.7 presents a user guide for working with the MTD and the currently featured plug-in monitoring tools. The following section describes the IPU's hardware interface more in detail.

### 11.3 The Integrated Probe Unit

In the following the IPU's integration with the target system is described, including its connections to the hardware RTK, and how it is accessed and controlled from a host computer system. The IPU is fully implemented in VHDL, and thus, is included in an RTL-description of the hardware that comprise the RTK.

#### 11.3.1 Entity interface

The IPU's signal interface is depicted in Figure 11.2. This *entity* view of the IPU shows a specific version that has a host port that implements the EPP (Enhanced Parallel Port [3]) protocol as the communication method in the Host Interface (see following section). In another implementation of the IPU, the host port is replaced with a USB interface [4]. The discussions throughout this report assumes, however, an EPP connection is provided as the host port. A description of the signals is given in Figure 11.3.

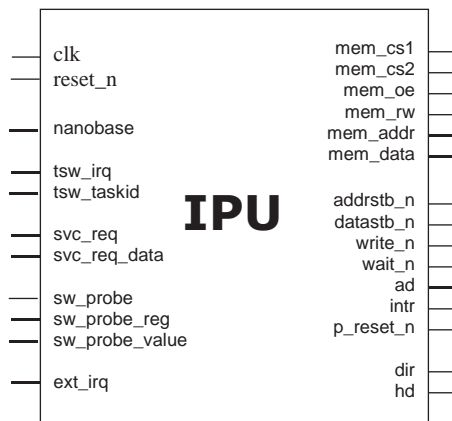


Figure 11.2: The IPU's entity interface

The signal *nanobase* is an input vector which sets the number of elapsed nanoseconds in a *clock period*. This is used by time-stamp module within the IPU. For example, if the clock speed is 10 MHz, then each clock period is 100 ns long. The vector may be hard-coded in the synthesised hardware, or more preferably, be programmable from software. The necessary logic for the nanobase input must be provided.

The *tsw\_irq* is a strobe signal that indicates the occurrence of a task-switch. The signal is active 'high' during the task-switch, and the ID of the new task should be provided using the *tsw\_taskid* vector. In a hardware RTK, the equivalence of the *tsw\_irq* and *tsw\_taskid* respectively are available between the scheduler module and the external bus-interface. In a multiprocessor version of the RTK, there exists one pair of *tsw\_irq* and *tsw\_taskid* for each CPU in the system. Hence, the *tsw\_irq* may be a vector with a length equal to the number of CPUs. The *tsw\_taskid* vector length is the number of CPUs multiplied with the number of bits required for the task-ID. For example, in a system with 3 CPUs and an RTK that supports 256 tasks, the *tsw\_irq* has a length of 3 and the *tsw\_taskid* has a length of  $3 * 8 = 24$  bits.

The *svc\_req* is also a strobe signal which, when 'high', indicates that a *service-call* was issued to the RTK, and the *svc\_data* is the vector that contains the actual parameters for that service-call. Like *tsw\_irq*, the *svc\_req* may also be a vector in a multiprocessor RTK. *svc\_data* is typically routed from a (32-bit wide) register within the RTK's bus-interface, and the *svc\_req* signal may be extracted also from the process that manages the bus-interface (i.e. the bus address decode logic).

The IPU's software probe registers are similar to the register provided for service-call management, i.e. software must write to a dedicated register in the RTK's interface. Therefore, the *sw\_probe* is a strobe signal that behaves equally the same as *svc\_req*. It is 'high' when a probe register is written to, and the written data must be provided with the *sw\_probe\_value* vector (32-bit). In order to differentiate between different types of software instrumentation, it is possible to support multiple software probe registers in the hardware interface to the IPU/RTK. Thus, the signal *sw\_probe\_reg* may be used to indicate *which* register was accessed. The necessary decode logic that handles this separation must be provided, e.g. by modifying the RTK's bus-interface, or by providing a separate bus-interface for the IPU.

The *ext\_irq* signal vector may be routed directly from the interrupt source signals that goes to the RTK. An interrupt is detected when one of the vector's signals goes 'high'. This signal must then be negated in order to be detected again (i.e. it must toggle for multiple interrupts).

The signals named *mem\_...* constitutes an interface to an external RAM-module which may be used to extend the internal FIFO buffer (see Section 11.3.2.1). The size and data widths of this memory interface may be adjusted using the IPU's *generic port configuration*.

The other signals conform to the EPP interface (see [3]).

Signal	Direction	Description
clk	IN	System clock
reset_n	IN	System reset
nanobase	IN	Time-base unit for time-stamp
tsw_irq	IN	High on task-switch
tsw_taskid	IN	ID of new task
svc_req	IN	High when service-call is asserted
svc_req_data	IN	Service call argument
sw_probe	IN	High if sw-probe register access
sw_probe_reg	IN	Sw-probe register select
sw_probe_value	IN	Sw-probe value
ext_irq	IN	External interrupts
mem_addr	OUT	Memory address
mem_cs1	OUT	Memory chip select Low-byte
mem_cs2	OUT	Memory chip select High-byte
mem_oe	OUT	Memory output enable
mem_rw	OUT	Memory read/write
mem_data	IN/OUT	Memory data input/output
addrstb_n	IN	EPP address strobe
datastb_n	IN	EPP data strobe
p_reset_n	IN	EPP reset
write_n	IN	EPP data direction
wait_n	OUT	EPP host acknowledge signal
ad	IN	EPP data input/output
intr	OUT	EPP interrupt to host signal
dir	OUT	For IEEE1284-buffer steering
hd	OUT	For IEEE1284-buffer steering

Figure 11.3: Signal description of the IPU entity interface

### 11.3.2 The Host Interface

The IPU's *Host Port* module provides a logical programming interface for communication with the host-based software (i.e. the MAMon Framework). The events collected by the IPU are sent via this interface, and vice versa, the interface is also used by the host software to control and supervise the IPU. Regardless of the type of Host Port that is used (EPP, USB, etc), the programming

interface is logically the same.

The programming interface consists of three 8-bit wide registers: 1) MONSR – contains various IPU status bits, 2) MONCR – a register that is used for controlling the IPU, and 3) MONEVT – the register that is used for reading the collected event data bytes. Due to the different types of Host Port modules that can be used, we do not go into the details on how these register are accessed from a host computer. Instead, we refer to the specific documents available for the Host Port communication [3, 4].

Following is a description of the MONSR and MONCR registers respectively. The MONEVT register is not described since it is mainly used for reading raw event data bytes (exceptions to this behaviour is mentioned below however).

#### 11.3.2.1 MONCR – Monitor Control Register

Acronym	R/W	Address	7-6	5	4-3	2	1	0
MONCR	R/W	\$01	Not Used	Extended FIFO	IRQ Mode	Error Detection	Stop Condition	Start

The register's address (\$01) is relative to a *base address* for the IPU (assuming the IPU is accessed as a memory-mapped device). Moreover, the register can be *read* from, and *written* to.

Bit-field descriptions:

**Start** Enables/Disables monitoring

0 = disabled [default]

1 = enabled

**Stop Condition** Tells the IPU to stop monitoring when the FIFO buffer is full.

0 = continuous monitoring

1 = stop when FIFO buffer is full [default]

**Error Detection** Enables *read error detection mode*. When this mode is set, each event data byte read from MONEVT has to be written back to MONEVT in order to be verified. If a mismatch between what is read from/written to MONEVT is detected the *Read Error* bit in MONSR will

be set. Also, in this mode, the event data pointer will only advance when correct data is written back.

0 = off [default]

1 = on

**IRQ Mode** Selects between polled event acquisition mode, and interrupted modes of event acquisition.

00 = off (polled mode) [default]

01 = new\_evt\_mode

10 = buf\_halffull\_mode

11 = buf\_full\_mode

**Extended FIFO** Enables/Disables external RAM to be used as an extension of the internal FIFO buffer

0 = disabled

1 = enabled [default]

**11.3.2.2 MONSR – Monitor Status Register**

Acronym	R/W	Address	7-5	4	3	2	1	0
MONSR	R	\$00	Not Used	IRQ	Buf Full	Buf HalfFull	New Event	Read Error

The register’s address (\$00) is relative to an assumed *base address* for the IPU. Moreover, the register can only be *read* from.

Bit-field descriptions:

**Read Error** Indicates that a read error has occurred. This bit is set if the data byte written to MONEVT does not match the data byte previously read from MONEVT. Only valid when error detection mode is enabled.

0 = no error

1 = error

**New Event** Indicates whether there are any event data bytes left to read from the FIFO buffer. This bit is effectively used to check for new events in the *polled* event acquisition mode (see *IRQ Mode* bits in MONCR).

0 = empty FIFO buffer

1 = (at least) one data byte is in the FIFO buffer

**Buf HalfFull** Indicates if the FIFO buffer is half-filled. Normally used together with the interrupt mode *buf\_halffull\_mode* (see MONCR).

0 = FIFO is not half-filled

1 = FIFO is half-filled

**Buf Full** Indicates if the FIFO buffer is full. Normally used together with the interrupt mode *buf\_full\_mode* (see MONCR).

0 = buffer is not full

1 = buffer is full

**IRQ** Indicates if the IPU generated an interrupt request. This bit is cleared only when the reason for the interrupt is negated.

0 = no interrupt is requested

1 = interrupt was requested

## 11.4 The MAMon Application Framework

This section explains how the Framework components are organised and function, specifically how the MTD should globally look and the features it should support. First, the communication between then MTD program and the hardware IPU is explained, followed by a short sketch of how the MTD is used.

### 11.4.1 Connection with hardware

The event data collected by the hardware can be transferred to a host machine via an EPP connection (or a USB connection as described in [4]). A library provides the access to the hardware using this connection. The monitoring process can be started and stopped from the MTD, and events collected by the IPU can be read. The MTD uses this interface to control the monitoring process and to store the collected events in a database. A technology called the Java Native Interface (JNI) is applied in order to access this library in the Java code of the MTD.

Collecting events from the hardware and directly writing them in the database would make the process dependent on the speed of the database. Therefore a medium is used. The medium chosen here is the so-called FIFO file, a special file type for such purposes. When both a reader and a writer have opened the file, data can be passed through it. Writing events into a FIFO file is no different from writing into a normal file. Similarly, the database command that reads from the FIFO file is the same as the database command for reading from a file. The reading process of the database will not finish until after the FIFO file has been explicitly closed by the writing party.

For the MTD program to control the database, the Java DataBase Connectivity (JDBC) is used. This enables access to a database from within a Java program using SQL queries. This same functionality is put in to use when the event data is required by the monitoring tools (i.e. the plug-ins) within the MTD.

A schematic view of the path event data follows is shown in Figure 11.4.

### 11.4.2 The main program

The main program consists of the MTD on which the plug-ins are available in smaller windows. In order to use these plug-ins, one should first create a new (or open an existing) *session* which stores the data and settings to re-use it later. Now the tools can be used on the data that was stored in the database or new data can be obtained from the hardware.

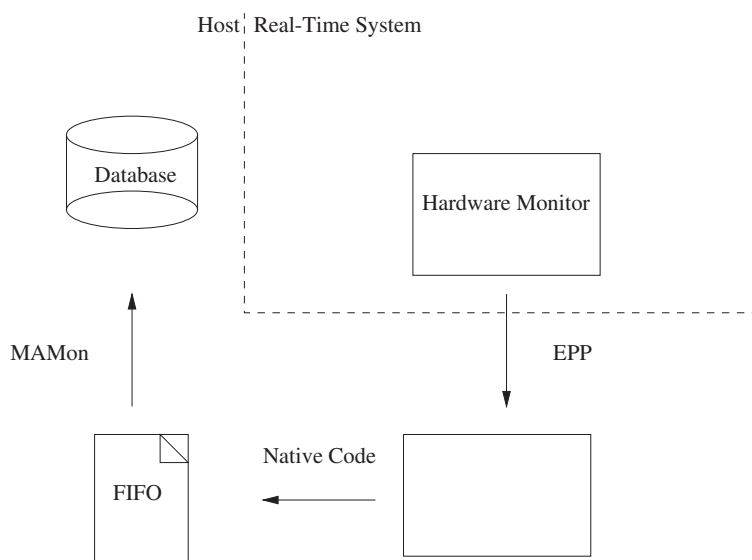


Figure 11.4: Schematic view of transport of event data

Starting the monitoring will remove any data that was previously stored in the session and the new event data is stored instead. Statistics on the number of events that has been transported are displayed while the monitoring process is active. It is possible to use all the plug-ins while monitoring.

The plug-ins that currently are present in the MTD are:

**Text plug-in** Can be used by the MTD program or other plug-ins to display texts, such as query results. The user is able to edit the text and save it to file.

**Event Query plug-in** A query-interface to select events the user is interested in.

**Event Viewer plug-in** Gives a graphical overview of the events that occurred over time.

**Chart plug-in** A tool that can display line charts, but without any specific charts implemented yet.



These plug-ins may cooperate with each other on a certain level. For example, the results of a query in the Event Query plug-in may be displayed using the Event Viewer to give the user an idea of the events' context.

## 11.5 Framework Software Architecture

In this section the architectural solutions applied in the Framework are described. These solutions are based on a set of design requirements which we think are important. The section concludes with a description of all packages and their classes.

### 11.5.1 Packages

Based on the different functionalities within the MTD, the following general division into packages are made:

**database** contains classes dealing with access to the database.

**event** contains classes for the representation of events loaded from the database.

**gui** contains classes dealing with graphical representation, except for the plug-ins.

**ipuBridge** contains classes dealing with communication with the IPU.

**plugIn** contains an interface for plug-ins, and the plug-ins' classes themselves.

**session** contains classes dealing with storing and retrieving session information.

### 11.5.2 Meeting requirements

In this section we present the design requirements for the Framework and the implications each requirement has on the Framework's architecture.

#### 11.5.2.1 Stability

To provide a stable working environment the MTD must be able to run independently from the hardware and the monitoring process. Therefore the program code that deals with the event collection is implemented in separate threads as thread classes. These are *IPUBridge*, *IPUThread* (both in package *ipuBridge*) and *DatabaseWriter* (in package *database*). See also Section 11.5.4.

#### 11.5.2.2 Extendability

As the name plug-in suggests, it should be possible to write a new one — or rewrite an existing one — and 'plug it in the MTD program', without modifying or even re-compiling the MTD. This means the MTD must be able to use

classes at run-time it did not know about at compile-time. The MTD program could be pointed to these classes by user input. Such a feature is available in Java and is called *dynamic extension* ([10]).

Using classes that are unknown at compile-time makes it impossible for the MTD program to use any of the plug-ins functionality unless it knows something about the plug-in. Therefore, all plug-ins should implement a common interface, so the MTD program can use the general methods defined in the interface, while the exact implementation of the plug-in is still free.

This general interface for plug-ins is in the class *PlugIn*, in the package `plug-in`.

### 11.5.2.3 Independence

Making the MTD's components independent means that it should be possible to change the implementation of a part of the MTD program or any external components, without influencing the rest of the MTD program. For example, addition of a new plug-in should not influence any other part of the MTD program. The same should hold for external factors influencing the MTD, such as the type of system that is monitored or the database that is used. This level of independence is present all over the MTD; different functionalities are separated from each other, and the graphical representations are separated from its actual implementation.

### 11.5.2.4 High-level Query Language

The plug-in called Event Query (class *EventQueryPlugIn*) provide a user interface for interactive querying of the event database. The user can make his own selection of events. Because no query language is used, the level of expression is not equal to that of f.e. SQL, but it is easier to use and fitted for this particular purpose. The results of a query can be displayed on screen, written to file or transported to plug-ins with graphical capabilities.

To make this querying tool more user-friendly, we want to use the real names of the events instead of having to use the corresponding numeric values (which is what is stored in the database). We would also like to use these names in other plug-ins. Stored in the database are three values for each event:

**type** the type of the event

**time** the time at which this event occurred

**parameters** a field containing values relevant for this type of event. Not the

entire space available is necessarily used.<sup>1</sup>

The values we would like to translate are the type and the parts of parameters that are defined for an event. For these translations, we create an *event definitions* file. This file contains:

- definitions of events, containing the name and number of the event, and the parts of parameters that are defined for this type.
- definitions of fields — parts of parameters — using the name and where in parameters (bit-values) the field is located.
- definitions of constants, a possible value a field can take. A constant is defined within the scope of an event type and a field.

This file is read by the class *EventDefinitions*. The same class then provides the rest of the MTD program with the translations. For example, the Event Query plug-in can request the names of the defined events, and the fields defined for these events. It can display these names to the user. When the user performs a query, the names can be translated to numbers again (using methods from *EventDefinitions*), so the database can be queried.

#### 11.5.2.5 Session

In a session, monitoring settings can be stored, along with settings by the various plug-ins. The settings are not stored on file, but in a database table. The user can only use the MTD to manage sessions. The MTD takes care that the database table and the session information are created and deleted at the same time.

The class *SessionManager* deals with saving/loading/updating/deleting a session (an object of type *Session*) in the database.

Because the sessions are stored in the database, it does not make sense to store *database* settings in the session. Instead, these settings — and other general program settings — are saved in a file. Its contents are read at start-up, and the settings can be changed from within the program. Since the settings involve a database-password, the settings are not written in a human-readable format, but as the serialised form of a class, *ProgramSettings*. An instance of this class is used by the MTD program to read and write the settings.

---

<sup>1</sup>Note that, for performance reasons, the parameter part of an event is actually stored in two table columns in the database.

### 11.5.2.6 User Interface

Most requirements for the MTD's user interface do not require any architectural features; they are mostly superficial. The only requirement that really influence the architecture is that the plug-ins should behave as in a *Multi-Document Interface* (MDI)<sup>2</sup>. Several classes together providing an MDI are presented in [8]. These classes are used in the MTD, be it in an adapted form.

### 11.5.2.7 Documentation

The only documentation that is directly connected with the MTD program is the on-line help. For this purpose the JavaHelp ([19]) is used. This standard Java add-on package separates the help-functionality from the rest of the MTD. Furthermore, it is very easy to extend the help documentation in case of, for example, a new plug-in.

## 11.5.3 Architecture Overview

In this section, an overview of Framework's software architecture is presented up to class level. A class diagram showing the relationships between all classes is illustrated in Figure 11.5. More specific documentation can be found in [7].

### 11.5.3.1 Package *gui*

The MTD is centred around the *gui* package. It is responsible for displaying the main program window, and linking all its buttons and menus to the correct functions provided by other packages, such as connection with the hardware in *ipuBridge*. Because the main window is contained in here, *gui* also controls the smaller plug-in windows, which reside in the main window.

#### **Class GUI**

This class manages all main graphical items. It is also the class where the start-up main method is located. All exceptions thrown outside the plug-ins are handled in here, and the graphical management of the plug-ins is regulated in here. All other classes in this package are just minor support classes.

#### **Class MDIDesktopManager**

Manager for the MTD program's *desktop* that supports MDI functionality for the plug-ins. Enables the use of a special window menu and a scrollable desktop. Adapted from [8].

---

<sup>2</sup>Examples of MDI's are word processors such as Word or WordPerfect.

**Class MDIDesktopPane**

Desktop that supports MDI functionality for the plug-ins. Adapted from [8].

**Class WindowMenu**

Menu for managing the plug-in windows. Contains functions for arranging the windows and the possibility to switch to a certain window. Adapted from [8].

**Class SplashScreen**

A splash screen window running in a thread, displaying an image. Used on start-up of the program.

**Class Gauge**

Graphical representation of a value on a logarithmic scale. Used to display the throughput (the number of events retrieved per second) when monitoring.

**11.5.3.2 Package database**

This package provides all necessary access to the database: reading and writing of data, but it can also deal with storing a session.

**Class DatabaseAccess**

This class provides connection to a database using JDBC. Once connected, queries may be performed, of which the results are returned. When finished querying, one can disconnect from the database. This is the only class that actually connects to a database, which means that all connections are made via an instance of this class.

**Class DatabaseReader**

Provides reading access to information in the database; it enables querying. When dealing with tables containing event data, the results are returned in the form of *Event* objects.

**Class SessionRetriever**

This class deals with a special table in the database used to store all session information. New sessions can be written, old sessions can be loaded or deleted.

**Class DatabaseWrite**

The storage of event data in the database is handled by this class, which runs as a thread.

**Class SubQuery**

A container class necessary to use a method from *DatabaseReader*.

**11.5.3.3 Package event**

Contains classes for the representation of events and for the translation of numeric values of the events.

**Class Event**

This class stores one (1) result row of a query on an event table. The result of a query is converted into *Event* instances for use by the plug-ins.

**Class EventDefinitions**

Reads name definitions of the events that occur in the hardware from file and of fields within these events and values that can be assigned to these fields. These names are used to display the events.

**Classes EventDef, FieldDef and ConstDef**

Container classes used by *EventDefinitions* to store information on the names of events, fields and constants.

**11.5.3.4 Package plugIn**

The code of the plug-ins is entirely located in the *plugIn* package. All plug-ins implement a common interface, necessary for *gui* to call general functionality such as displaying the plug-in.

**Interface PlugIn**

Interface which all plug-ins must implement in order to be available in the program.

**Plug-in classes**

All the classes of the plug-ins are also within the *plugIn* package. The main class of each plug-in implements *PlugIn*. These classes all have names ending on *PlugIn*. The other classes in these packages are container classes, or auxiliary classes for the plug-ins.

#### **11.5.3.5 Package ipuBridge**

Connection to the IPU hardware for reading and storing the events is provided here.

##### **Class IPUBridge**

Manages the retrieval of the event data from the hardware, using two threads, one of type *IPUThread*, and one of type *DatabaseWriter*.

##### **Class IPUThread**

This thread class communicates, via EPP, with the IPU. It reads event data, and makes it available for storage in the database. Part of the actual implementation is not done in Java, but in C. Access to this code is provided through JNI.

#### **11.5.3.6 Package session**

Container classes for sessions (containing session specific settings) and general program settings.

##### **Class Session**

Contains information on a session: specific settings for that session from the main program and from the plug-ins. A session belongs to a table in the database.

##### **Class ProgramSettings**

Contains the general settings of the program regarding among others database settings.

#### **11.5.4 The retrieval mechanism**

Although the MTD is a large program, most of its code is easy to understand. The main exception is the code that deals with the retrieval of events from the hardware and storing them in the database. In this section that mechanism is explained in more detail.

##### **11.5.4.1 Threads**

The monitoring process consists of the two tasks that need to be executed concurrently: events must be collected from the IPU (1) and events must be stored in the database (2). On top of this, we want the MTD to be able to perform other operations too when the program is busy monitoring. To achieve this goal, the MTD program uses the Java thread mechanism.



The classes *IPUBridge* and *DatabaseWriter* are both defined as threads. Initially, these threads are “suspended”, which means they are inactive. A thread can be suspended by issuing a `wait()`<sup>3</sup> command. When the user starts the monitoring process, the threads can be “awakened” using the `notify()` command.

Listing 11.1 schematically shows the program code dealing with the starting and stopping of the monitoring process. When the start button is pressed, `startRetrieval()` is called, and stopping the monitoring results in a call of `stopRetrieval()`. Note that this code is not the same as the actual code in the program; it is a simplified version.

Upon start we notify both threads and their suspension ends. Then, *IPUThread* resets the hardware, opens the FIFO and start reading events from the hardware. *DatabaseWriter* will have the database perform a query to insert the data from the FIFO.

When the monitoring process is stopped by the user, *IPUThread* stops executing `readEvents`, closes the FIFO and goes back into suspension. In the *DatabaseWriter*, the query will — eventually — end since the FIFO has been closed. Then, the *DatabaseWriter* gets suspended, too.

Listing 11.1: Starting and stopping

```
// IPUBridge
public void startRetrieval(){
    ipuThread.stopped = false;
    synchronized (ipuThread){
        ipuThread.notify();
    }
    synchronized (databaseWriter){
        databaseWriter.notify();
    }
}

public void stopRetrieval(){
    ipuThread.stopped = true;
}
...
// IPUThread
```

<sup>3</sup>A `wait()` command should be used within a `synchronized`-clause. This also holds for the `notify()` command.

```
public void run() {
    while (true) do {
        synchronized (this){
            wait ();
        }
        resetHardware ();
        openFIFO ();
        while (!stopped) do {
            readEvents ();
        }
        closeFIFO ();
    }
}
...
// DatabaseWriter
public void run() {
    while (true) do {
        synchronized (this){
            wait ();
        }
        PerformInsertQuery ();
    }
}
```

#### 11.5.4.2 Updating

When monitoring, it can be desirable to use the most recent data in the plugins without stopping the monitoring process. However, we can only obtain the latest data from the database after the insert query has ended. To achieve this, we must close the FIFO and therefore temporarily suspend the reading of events from the hardware. We wish to continue these processes as soon as possible, and therefore can skip the waiting.

However, when the waiting is skipped, the situation may occur that the insertion query does not finish. The scheduling of the threads may be such that the closure and opening of the FIFO occur before the database thread can exit the query. This can be solved by having *IPUThread* wait, being notified by *DatabaseWriter* when it has finished its query. This means, however, that we need to synchronise the behaviour of *DatabaseWriter* and *IPUThread*; we cannot have the *DatabaseWriter* notifying when the *IPUThread* is not yet waiting

(deadlock). This is achieved by including the `closeFIFO();` statement in the `synchronized-block` around `wait()`.

Finally, it is also necessary to prevent the plug-in that called for the update from accessing the database before the insert query has ended. This is realised by suspending the main program. Before a new insertion query starts, the main program is notified by *DatabaseWriter*.

Listing 11.2 shows the code for interrupting the retrieval process in order to update the database.

Listing 11.2: Updating

```
// IPUBridge
public void updateDatabase () {
    databaseWriter . updating = true ;
    ipuThread . stopped = true ;
    synchronized ( this ) {
        wait () ;
    }
}
...
// IPUThread
public void run () {
    synchronized ( this ) {
        wait () ;
    }
    while ( true ) do {
        resetHardware () ;
        openFIFO () ;
        while ( ! stopped ) do {
            readEvents () ;
        }
        synchronized ( this ) {
            closeFIFO () ;
            wait () ;
        }
    }
}
...
// DatabaseWriter
```

```
public void run() {  
    while (true) do {  
        if (updating) {  
            synchronized (ipuThread) {  
                notify ();  
            }  
            synchronized (ipuBridge) {  
                notify ();  
            }  
            updating = false ;  
        } else {  
            wait ();  
        }  
        PerformInsertQuery ();  
    }  
}
```

## 11.6 Framework Programmer's Guide

### 11.6.1 General

In this section, some general information for program development in the Framework is presented; which files and directories are present in the MTD program's file directory, how to compile and run the MTD, how to generate the Javadoc documentation.

#### 11.6.1.1 Files and directories

The MTD file directory will contain the following sub-directories. All of them are required by the MTD, with the exception of the `src` directory, which is only necessary for compiling the MTD.

**classes** contains all the class files, including a native shared library called *IPUThread.c*, which handles the communication with the IPU hardware. The class diagram presented in Figure 11.5 gives an overview of all the Java classes in the Framework. Each class is represented by a box. If a class is associated with another class, they are connected. The class at the end of the connection with a range is referred to by the class at the other end of the connection. The number of instances to which is referenced is indicated by the range, where `x..` means `x` or more (no maximum).

**documentation** contains the documentation generated by javadoc.

**help** contains the help files used in the program.

**images** contains the images and icons that are used in the program

**src** contains the source code for the program, including the source of the shared library (in the *ipuBridge* package directory), and the JDBC driver file.

In the MAMon installation directory a number of files are also present for use by the MTD program.

**event.def** contains name definitions of events. Required to run the MTD program.

**settings.dat** contains the general settings of the MTD. Required to run the MTD program.

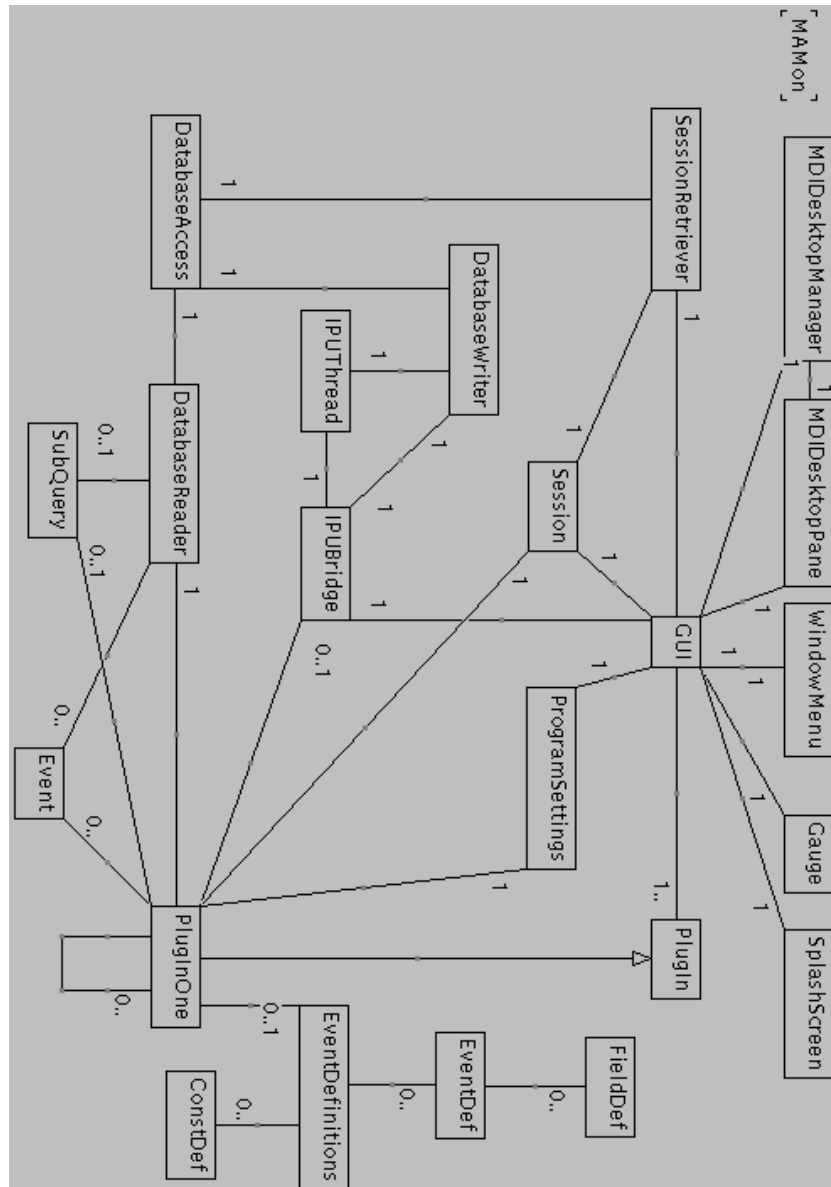


Figure 11.5: Class diagram

**file** the file used by the MTD to store data in the database. Required by the MTD program, but location and name are not fixed.

Besides these, there are also some shell scripts in the MAMon directory. Their use is explained in the coming sections.

### 11.6.1.2 *Compiling and running*

To compile or run the program, include the following in the classpath:

1. the directory where the classes are stored (classes)
2. the jar-file for JavaHelp `src/jhall.jar`
3. the jar-file for JCChart `src/jcchart451K.jar`
4. the jar-file for the JDBC driver `src/mm.mysql-2.0.4-bin.jar` (file name changes when using another DMBS, see Section 11.6.4).

When compiling the entire program, compile all the `.java` files in all package directories. When running the MTD program, run `gui.GUI`, that is where the `main()` method is located. Shell scripts called `run` and `comp` take care of running and compiling the program.

To compile the `IPUThread` library, go to `src/ipuBridge`, that is where the C-files `IPUThread.c`, `mamon.c` and `mamon_dev.c` are located. Shell `ccomp` compiles these files and places the resulting library in the `classes` directory. For more information on how to compile this library, see the JNI tutorial ([18]).

### 11.6.1.3 *Generating documentation*

The command `doc` generates the API documentation for MAMon using Javadoc. It is basically called with a list of all packages that should be included in the documentation. The following options are included, too:

- `-private` Includes documentation for private methods and fields.
- `-sourcepath src` The source directory.
- `-d documentation` The output directory of the HTML files.

In case more options are required, read [16].

The command `doctex` generates API documentation for the Framework with Javadoc, but the output is in the form of a `TEX`-document instead of HTML. For this, a so-called doclet is used. To use Javadoc with this doclet, use the following options:

- `-doclet com.c2_tech.doclets.TexDoclet` The name of the doclet.
- `-docletpath doclet.jar` The location of the doclet code.
- `-output javadoc.tex` The name of the output file.

More options for this doclet are discussed in [17].

### 11.6.2 Adding a new plug-in

To add a plug-in the main class of this new plug-in should implement an interface, *PlugIn*. After the newly created plug-in has been compiled (put the class-files in the same directory as the other plug-ins are located), you can use the plug-in by adding its name as an argument when starting the MTD program.

The following section gives guidelines and help on how the plug-in's interface should be implemented and how to use certain facilities the program offers.

#### 11.6.2.1 Interface

The interface *PlugIn* defines a number of methods that should be implemented for the plug-in to function correctly within the program, and a number of methods that can be implemented if required.

In order to make the plug-in visible in the main window, the method `getPlugInFrame()` should return a *JInternalFrame*. The easiest way to achieve this is to have the plug-in class to extend *JInternalFrame*. `getPlugInFrame()` can then return `this`, and the plug-in will be made visible in the correct way. On the *JInternalFrame*, a user interface can be built using *Swing* components. Take a look at Sun's *Swing* tutorial ([15]) if for how to make a graphical user interface.

Other methods that are necessary are `getPlugInName()`, which returns the name of the plug-in, and `getPlugInMenu()`. The contents of the menu of the plug-in are not restricted, but for conformity with other plug-ins the menu should at least include the items *Hide* and *Show*. Also make sure that closing the plug-in window will lead to hiding of the window, not closing, because a plug-in cannot be restarted from within the program. For an example on how to implement this exactly, check the source of an existing plug-in, such as the *Text* plug-in.

The size and location of each plug-in are stored in the program's general settings. Access to the *ProgramSettings* object is possible in the method `set-`



`ProgramSettings()`. This method is called by the main program, with the *ProgramSettings* object as a parameter. In the `setProgramSettings()` method, the plug-in can retrieve the size and location stored in this object by calling `getPlugInSettings()` with the name of the plug-in as an argument. Note that when the plug-in runs for the first time, there are no settings available it, so default values should be assumed for this case.

The final method that should be implemented for correct functioning is `cleanUp()`. In this method, the plug-in must prepare for a clean exit. Usually, this involves releasing its database connection (see section 11.6.2.2) and disposing of its *JInternalFrame*. If it uses any other resources, be sure to clean them up as well.

All other methods in the interface do not need to be implemented. However, to make use of certain functions, they will have to be implemented. For more information on these methods, see the remaining of this section.

#### **11.6.2.2 Accessing the database**

In order to access the database, an object of type *DatabaseReader* is from the package *database* is needed. This object should be constructed using information from the *ProgramSettings*, so the best place to initialise an instance of *DatabaseReader* is in the method `setProgramSettings()`. When the program exits and calls the plug-in's `cleanUp()` method, call the *DatabaseReader*'s `cleanUp()` method in order to release the connection with the database.

#### **Performing a query**

In case the query you wish to perform in the plug-in returns events, you can use either `eventQuery()` or `eventQuerySQL()`. The name of the table on which to perform this query is the same as the name of the current session (see section 11.6.2.3 on how to access that information). Neither of these methods will return any values; instead you must use the methods `moreEvents()` and `getNextEvent()`. `moreEvents()` indicates whether there are any results left that haven't been returned yet, and `getNextEvent()` returns the next event.

Whenever possible, you should use `eventQuery()`, and not `eventQuerySQL()`. Only use the latter one in case you can not express the query using `eventQuery()`. This is because you will need to enter literal SQL when you use `eventQuerySQL()`, and this may cause the plug-in to work incorrectly when the database component is modified. See section 11.6.2.2 for how to use `eventQuery()`.

In case the query does not involve events, for example, if you want to know the number of events that is stored in a particular table, you can use

querySQL(). This method returns an object of type *ResultSet*. See the documentation on JDBC for how to obtain information from it ([14]).

If you want a query to use the latest data, even when the MTD is monitoring, call the method `updateDatabase()` from *IPUBridge*. This will cause the monitoring process (when active) to stop momentarily, so that recent data will be available from the database for the plug-in. A reference to the instance of *IPUBridge* can be obtained from the interface method `setIPUBridge()`.

#### ***Using the eventQuery method***

The `eventQuery()` method can query for events in an event table, although the possibilities are less expressive than SQL. You can restrict the events returned in the following ways:

- *By time.* Use parameters `timeAfter` and `timeBefore` to specify the times (in nanoseconds) after or before which the events should take place (non-inclusive).
- *By number.* Limit the number of events returned is possible in three different ways. Returning the first `limitFirst` events (type 0), the last `limitFirst` events (type 2), or events `limitFirst` to `limitLast` (type 1). The type of limitation is defined using `limitType`.
- *By the values of the event.* The events returned will satisfy all the *SubQuery* objects passed as a parameter. Details on *SubQuery* are given below.

For all parameters holds that a value of -1 means that particular parameters will not be considered in the query.

Within a *SubQuery* events can be required to be of a specific type, and you can require a part of the `parameters`-field of an event to be equal (or not equal) to a certain value. All these requirements have to be entered as a parameter when creating a new *SubQuery* object. To get the correct bit-values of a specific parameter field, you can use the *EventDefinitions* (see section 11.6.2.4). This is preferred above including the values statically in the code, since that will make the plug-in useless when a change occurs in the way events are defined.

#### ***11.6.2.3 Sessions***

If there are settings in the plug-in that are defined by the user, you may want to store them in the current session, so that the same settings are used when the

same session is re-opened later. You also need sessions to obtain the name of the table in case the plug-in performs queries on the database.

### ***Saving information***

A *Session* object allows each plug-in to store one *Object*. In case you wish to store more than one value, you should use a container class for this. This can be of a type that is already available in Java, such as an array, or *Vector* (from the *java.util* package). Or you can create your own class in which all the values are put together.

Storing of the information you wish to save must occur in the `savePlugIn()` method defined by the *PlugIn* interface, where *Session*'s `setPlugInSettings()` can be used. Besides the object to be saved, that method also requires a unique name which should be used later to get the stored information back. The most straightforward name to use is the name of the plug-in; in this way, there will be no mix-up of settings.

### ***Restoring information***

Whenever a new session is loaded, the method `loadSession()` is called for all plug-ins. At this moment, any stored information from that session can be accessed using the `getPlugInSettings()` method with the name used when the information was stored. This information will be returned in the form of an object.

Be sure to implement `loadSession()` so that it can also handle sessions in which no information was stored for the plug-in; assume default values for these settings in these cases.

From the *Session* object received in the `loadSession()` method, you can also get the name to use in a database query. Use `getName()` for this purpose.

#### ***11.6.2.4 Event names***

Information regarding events, obtained from the database, is not really suitable for displaying. The *Event* class provides a basic method which will return a printed string representing that event. If you want to use the actual names of events, fields and constants otherwise, use the class *EventDefinitions*. An object of this type can translate the numerical values returned by the database into the names of the events they represent, and vice versa.

An object containing this information is passed through the method `setEventDefinitions()` from the common plug-in interface.

### ***11.6.2.5 Communication with other plug-ins***

Sometimes, it is useful to use functionality of other plug-ins. But because the program does not know if a certain plug-in is available in the program or not, it is not possible to refer to an instance of, for example, the Event Viewer from the Event Query plug-in. However, there is a way in which plug-ins can communicate with each other.

#### ***Using functionality of other plug-ins***

The method `setOtherPlugIn()` from the common interface, called by the main program, provides an array of all the plug-ins that are available in this run of the MTD. They are only accessible as *PlugIn* objects, so only the methods defined in that interface can be called. Two of these methods are currently defined for use by other plug-ins: `writeMessage()` and `displayTimeStamp()`. `writeMessage()` displays a message (string) on screen, while `displayTimeStamp()` (graphically) highlights a time. These methods need not be implemented by any of the available plug-ins, so if you want to use one of them, call it for each plug-in from the array (note that the array includes a reference to your plug-in, too).

If these methods do not offer the functionality required, you could consider extending the interface *PlugIn* with new methods. If you do so, you must adapt all existing plug-ins so they support this new method, even if they do not implement it.

#### ***Offering functionality to other plug-ins***

If you want to make some of your functionality available to other plug-ins, all you need to do is implement the methods mentioned in the previous section so they perform the actions you desire.

As examples, the Text plug-in in the MTD implements the `writeMessage()` method; it simply writes the message in its text area. The Event Viewer moves its time marker to the time specified by the caller of `displayTimeStamp()`. The plug-in Event Query makes use of both these features to display the results of a query posed by the user.

### ***11.6.2.6 Help files***

Each plug-in should also include help files. When the MTD is running, these will be visible together with that of the MTD program. When the plug-in is not running, its help pages are not available.

A plug-in's help files should be written for JavaHelp (see [19] for how to write such files) and located in the directory `help/plugin/YourPlugIn`.

The program will give produce errors when the help files are not found.

### *11.6.3 Using the event definitions file*

As explained in section 11.6.2.4, the file with event definitions can be used to display the event data in a readable way. This section defines how the file containing this data is built up.

#### *11.6.3.1 Structure of the file*

The event definitions file is default named `event.def`. In this file you can define event types by their name and the different parts of the parameters field that represent a field. A field is defined by a name and the first and last bit it occupies in the 5 byte parameter field (so bit values range from 0 to 39). A special event named "all" is defined by default. Use this type of event to look for a specific field within several events. For example, in both the `TaskSwitch` and the `ServiceCall` event, the `CPU` field is defined at the same location.

Apart from defining events, constants can also be defined. These are possible values which a field of an event can contain. For example, the value 16 represents a `ServiceCall` event of Type (field) `Thread_Create`.

#### *11.6.3.2 Grammar*

The grammar for the event definitions file, in Extended Backus-Naur Form, is:

```
EventDefinitions ::= { <Event> | <Constant> }
<Event> ::= 'event' <EventName> <Value> '{' {<Field>} '}'
<Field> ::= <FieldName> <Bit> <Bit>
<Constant> ::= 'const' <EventName> <FieldName>
               <ConstantName> <Value>
<Bit> ::= 0..39
<EventName> ::= <Name>
<FieldName> ::= <Name>
<ConstantName> ::= <Name>
<Name> ::= 'A'..'z'
<Value> ::= 0..
```

### *11.6.4 Changing the DBMS*

When you change the DBMS used in the Framework for storing events, take care of the following issues.

First, get a JDBC driver for the DBMS you will use. Make sure the driver files are in the class-path. Then, open the program settings dialog in the main program and change the name of the JDBC driver to the new driver. You need give the full name of the package. The program needs to be restarted for the new driver to take effect. A list of drivers is available at [13].

With the new driver, almost all queries can be performed without change. There are two exceptions. The query that is used for storing new events in the database is dependent on the DBMS that is currently used, viz. MySQL. Reason is that insertion queries from file are not standard ANSI-SQL. However, most DBMSs offer such functionality, but with different syntax. The Framework offers room to implement another way to perform the insertion query. This new approach should be implemented in the `run()` method of the class *DatabaseWriter*. There is a switch-statement, which choose a user-defined method of inserting. Currently, only option 0 is implemented. The option selected can be changed in the MTD program's settings dialog.

A second place where the Framework depends on MySQL is in the class *SessionManager*. The method `sessionTableAvailable()` uses the MySQL specific command "SHOW TABLES", to give a list of all tables available. A similar command is usually available in other DBMSs, change this command so it works with the new choice of database.

## 11.6.5 Specific plug-ins

### 11.6.5.1 Event Viewer

The event viewer displays icons for events other than taskswitches. For service calls, there are even different icons for different service call types. To change these icons, do the following:

- Put the images you want to use in the *images* directory.
- In the method `loadImages()` in the *EventViewerEvent* (defined in the *EventViewerPlugIn* file), add the images for service call types to the `svcImages` array, and the images for event types to `eventImages`. Make sure the indexes used for the arrays are the same as the numbers of the event types/service call type. You can use the event definitions file for that purpose, too.
- Add the array positions just defined to the instance of *MediaTracker* defined in the same `loadImages()` method. This will ensure that the images are loaded only once.

### 11.6.5.2 Chart Viewer

The Chart plug-in has the possibility to add new types of charts. This section describes how to write a new chart type, and how to include it in the plug-in. For displaying the chart, the chart part of the JClass package is used (see [12]).

#### **Implementing a new chart type**

The new chart type you make must implement the interface *ChartType*. This interface has five methods.

First of all, decide whether chart you want to draw should be updated at a fixed interval. If so, the method *isTimedUpdate()* should return true, and *getUpdateTime* should return the interval (in milliseconds) between two updates.

The method *setChart()* provides the object on which the chart will be displayed. Using this object, of type *JCChart*, you can adapt the visual settings of the chart. For example, you can display a legenda, or set the colours to be used (see [12]).

Next, you can (not obligatory) define a dialog which will pop up when the user requests the chart to be displayed in *displaySettingsWindow()*. Here you can ask the user for input, for example the time range the chart will cover.

The most important method is *update()*. This method should return an object of type *ChartDataModel*. A very basic implementation is available under the name *DataSource*, in which the x-values and a number of corresponding y-values (so you can display multiple lines at the same time) should be entered. Now, all you need to do is calculate the values to display. For this purpose, access to the database and the currently opened session are present. For more information on how to use these, see Section 11.6.2.

If you require a more sophisticated *ChartDataModel* for your chart, you can implement your own (see [12]).

#### **Including the new chart type**

Add the code for your chart type to the same source file as the plug-in (*ChartPlugIn.java*). Then, add the name of your chart type to the array *chartTypeNames* in the *ChartPlugIn* class and add an instance of your chart type to the array *chartTypes*. Be sure that the indices are the same in both arrays.

Now, recompile the Chart plug-in, and the new chart type will be available when you restart the program.

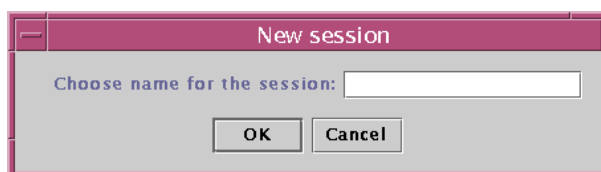
## 11.7 MAMon Tool Desktop User Guide

### 11.7.1 Quick start

This section explains in five short steps how you can use MTD. You will open a session, collect monitoring data from the hardware, analyse this data, and exit the program.

#### **Create a session**

Before you can start doing any monitoring, you need to create a new session. To do this select `Session | Open` or press `CTRL+O`. You will see the following window appearing:



Enter a name for your session and press `OK`. Now you have created a new session.

#### **Start monitoring**

Now, you can start to collect events from the hardware. Make sure a connection with the target system exists and that a program is running (or is ready to run). On the toolbar of the MTD, you will see the following buttons.



Press the leftmost button, i.e. the `START` button. Monitoring will start now. The left bottom of the program, you should see the "Status" caption change from "idle" to "monitoring". You will also notice other values on the status bar in the bottom of the screen will change.

#### **Stop monitoring**

When you have monitored long enough, you can stop the process by pressing the `STOP` button, which is the rightmost button on the toolbar above the program's desktop. The monitoring will now end. The "Status" caption in the bottom left corner of the screen will display "idle" again.



### *Analysing data*

Now you have collected event data, you can start to analyse it using one of the plug-ins. The plug-ins are the smaller windows resident on the MTD.

### *Quit program*

When you are done analysing and monitoring, you can exit the MTD by selecting `Session|Exit` or pressing `ALT+F4`. You can not exit as long as you haven't stopped the monitoring. The session that was open when you pressed exit is automatically saved.

## *11.7.2 Reference guide*

In this reference guide, you can find detailed information regarding the following topics:

**Events** - MAMon monitors events. What are they and how to use them?

**Event time format** - All throughout the MTD a specific time format is used. How does it work?

**Menu** - What do all the items in the different menus do?

**Monitoring** - How do I operate the monitoring process?

**Plug-ins** - What are plug-ins and how can I manipulate their windows?

**Sessions** - How to deal with sessions?

**Settings** - How to adjust settings, and what do they mean?

### *11.7.2.1 Event format*

The data that a monitoring session with the MTD delivers consists of events. An event consists of three parts, a type, a timestamp and parameters. In the plug-ins, you will see different representations of events. The representations are based on the three basic parts of an event, which are all stored as numerical values. However, the MTD is also able to assign names to those numbers.

#### *Type*

Each event has a type, represented by a number. Examples of types can be task scheduling events, communications events (both events generated by the hardware) or software triggered events.

***Time***

Each event is accompanied by a timestamp. This denotes, in nanoseconds, the time at which this particular event occurred. For display, a special format can be used.

***Parameters***

The final component of an event is a field of parameters. The exact contents of this field are different for each type of event. For some events, only a small part of the parameters contains sensible data such as the CPU on which the event occurred, while other types may occupy the entire field, such as the contents of a register.

A plug-in that displays an event will only display the parts that are relevant for that event.

***Definitions of names***

By using the event definitions file (`event.def`), you can define names to be used within the program. In there, names for event types are defined (for example, event type 1 corresponds to a task-switch event). Then, the fields within the parameters are defined (for example, event type 1 contains two fields named CPU and TaskID). Finally, you can define constants. For example, within the context of event type 1, the value 0 for the field TaskID means the *idle task* was executing. Instead of displaying a zero, you can display a `IdleTask` here.

You can enable the use of names in the Program Settings, where you can also set other issues regarding the display of events.

For information on how to edit the event definitions file, see the Programmer's Guide.

***11.7.2.2 Event time format***

In several places in the program and in plug-ins you will encounter the event time: when monitoring, when querying for events, when viewing event representations. The time format used for events is the same throughout the MTD. It is constructed as follows:

```
hours:minutes:seconds,milliseconds,microseconds,nanoseconds
```

As an example, the following represents 1 hour, 22 minutes, 44 seconds, 0 milliseconds, 85 microseconds and 3 nanoseconds (or 10000003 nanoseconds):

01:22:44,100,000,003

When times are printed, they are displayed like the one above, including the full number of zeroes. However, if you need to type in a time yourself, you can use shorter forms. The following can be left out when typing:

- The parts of at the start of the time that are not used. For example, when typing 1 second, the hours and minutes fields are be zero. Instead of 00:00:01,000,000,000 you can type 01,000,000,000.
- Leading zeroes. For example, when typing 1 second, instead of 01,000,000,000 you can type 1,0,0,0.

### **11.7.2.3 Menus**

The MTD program has the following five menus:

- Session - control sessions
- Edit - change program settings or edit text
- Tools - access to the plug-in functionality
- Window - adjust plug-ins and the general view
- Help - help and information regarding the program

#### ***Session menu***

The session menu has five items, with the following functionality:

- New - create a new session
- Open - open a stored session for usage
- Close - close the currently opened session
- Delete - delete a stored session
- Exit - quit the MTD

#### ***Edit menu***

The edit menu has five items, with the following functionality:

- Copy - copy the selected text to the clipboard
- Cut - cut the selected text and copy it to the clipboard
- Paste - paste the contents of the clipboard to the specified location
- Program Settings - adjust general settings regarding the program
- Monitor Settings - adjust settings regarding the monitoring process

#### ***Tools menu***

The tools menu has as many items as there are plug-ins available. The items each plug-in offers is different, but typically there are at least two items present, viz.:

- Show - makes the plug-in visible on the desktop
- Hide - makes the plug-in invisible on the desktop

#### ***Window menu***

The windows menu has 4 items plus as many items as there are plug-ins available. The four items are:

- Tile - orders the plug-in windows by filling the desktop
- Cascade - orders the plug-in windows cascading from the top left corner of the desktop
- Cross-platform 'Look & Feel' - the program uses the Java look and feel
- Platform specific 'Look & Feel' - the program uses the look and feel of the local operating system

The remaining items give a list of all the plug-ins available. The item that is checked is the plug-in that is currently in front at the desktop. If you select another, this plug-in will be moved to the front. In case the plug-in is hidden, it will be made visible, too.

#### ***Help menu***

The help menu has only 2 items:

- Help - displays the help environment
- About - displays information on this program

#### ***11.7.2.4 Monitoring***

Before you can use the plug-ins to analyse data, you first need to collect data. This process, monitoring, is controlled from the MTD too. In order to start monitoring, you first need to have opened or created a session.

##### ***Control***

The control panel for monitoring consists of three buttons, START, PAUSE and STOP.

START begins the monitoring process. In case you have any event data stored in the session, this data is removed, and the new data will be stored instead. Before starting, the hardware is reset.

When you are monitoring you can PAUSE the process, which cause the MTD to stop monitoring. When the PAUSE button is pressed again, the process is resumed. The data is added to the data stored before the pause, and the hardware is not reset. Resuming is also possible by pressing the START button.

To stop the monitoring process, press STOP. Monitoring can now only be started again using the START button.

##### ***Status***

While you are monitoring, you can see information on the process in the status bar. From left to right, you see the following:

- Status - Displays "Idle" when not monitoring, "Monitoring" when monitoring, and "Paused" when monitoring is paused.
- Error Detection - Displays "Err" when an error occurred in the communication. This only works when Error Detection is selected in the Monitor Settings.
- Throughput - Displays the number of events per second that is stored in the database right now.
- Counter - Displays the total number of events that was stored in the database since monitoring was started. This information is still available after the monitoring has stopped.
- Time - Displays the elapsed time on the monitored system since monitoring was started. This information is still available after the monitoring

has stopped.

- Session - Displays the session that is currently opened.

#### **11.7.2.5 Plug-ins**

To analyse the data you have retrieved by monitoring, the MTD provides a set of tools called *plug-ins*. The number and kind of plug-ins that is available within the program is not fixed, it depends on the choice of the user and the availability of plug-ins. The plug-ins that are available in a run of the MTD are all shown as small windows on the desktop of the main program. You can also see a list of their names by using either the `Tools` or the `Window` menu.

#### **Choosing which plug-ins to use**

You can choose which plug-ins you want to be present in a run of the MTD. If you just start the MTD, the program will run with a set of plug-ins that has been set as default. You can define exactly which plug-ins are present by adding them as an argument when starting the program. For example, the program is normally started as follows, and the default plug-ins are used:

```
java -classpath yourClassPath gui.GUI }
```

If you start the program as follows, only the plug-ins called Event Viewer and Event Query will be available:

```
java -classpath yourClassPath gui.GUI EventViewerPlugIn  
      EventQueryPlugIn
```

The names you provide in this way must be equal to the names of the class-files for these plug-ins. These class files must be located in the directory `MAMon/classes/plugIn`.

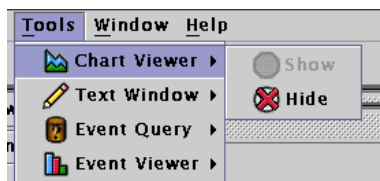
#### **Size and position**

In the program you can change the size or position of a window in the way the windows are resized or repositioned in your operating system. To order the plug-in windows, you may also use the functions *tile* (`Window|Tile`) and *cascade* (`Window|Cascade`). *Tile* will attempt to show all the plug-ins at the same time while filling as much of the desktop as possible. *Cascade* will resize the windows to their original sizes and cascade them, starting in the upper left corner of the desktop.

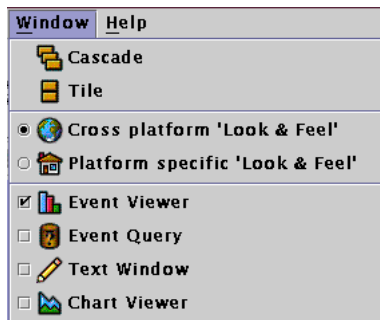
The sizes and positions of the plug-ins are stored when you close the program, so they will appear in the same way when you restart the program.

**Showing and hiding**

When you are running the program, it is impossible to close a plug-in. However, if you do not wish to see it on the desktop, you can hide it. There are two ways to achieve this. The first one is by using the window closing feature of your operating system. The second method is to select the plug-in you wish to close from the **Tools** menu. Then select the option **Hide** (see image below).



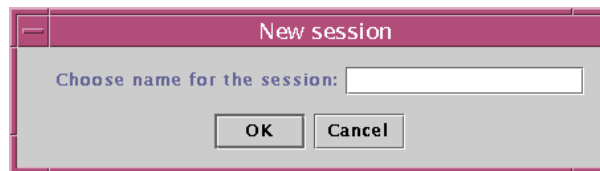
If you want to make hidden plug-ins visible again, you can do this by choosing that plug-in's name from the **Tools** menu and selecting **Show**. Alternatively, you can go to the **Window** menu, and select the plug-in you want to make visible. This option will bring a plug-in to the front, whether it was hidden or not (see image below).

**11.7.2.6 Sessions**

Before you can start monitoring the system with the MTD you need to create a *session*. A session will store the data you have monitored so you can analyse it later using the plug-ins. A session also stores the monitor settings and plug-in settings you used.

***Creating a new session***

You can create a new session using `Session | New` or `CTRL+N`. You will see a dialog where you are asked to type in a name for the new session (see image below). In case no session with the name you provided already exists, a new session with the name is created.

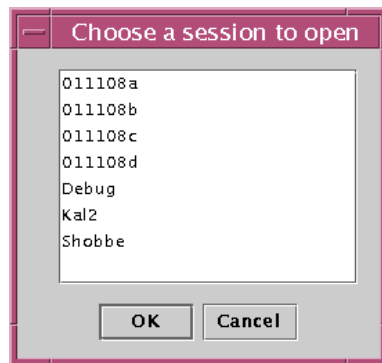
***Closing the current session***

When you do no longer wish to use the session you are using at the moment, you can close it using `Session | Close` or `CTRL+F4`. The monitor and plugin settings as they are will be saved in that session, and any data you have collected is stored with the session. Before you can start monitoring again, you will need to create a new session or open a session you previously created.

***Opening a previously stored session***

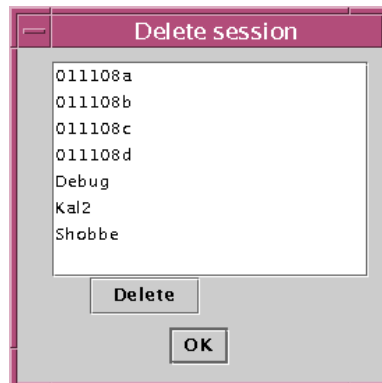
When you want to use the settings and/or data of a session that you stored earlier, you can open this session using `Session | Open` or `CTRL+O`. You will see a list of all the sessions that are stored (see image below). Select the session you wish to open and click OK. The session you selected will now open, and the settings will be adapted to match the settings stored in this session. If you had a session openend before you chose to open another one, that session will be saved before second one is opened.





#### ***Deleting a previously stored session***

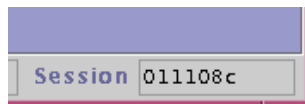
When you no longer wish to keep a session stored, you can delete it using `Session | Delete` or `CTRL+D`. You will see a list of all the session that are stored (see image below). To delete, select the session you want to remove and press the Delete button. If you want to delete more sessions, repeat this procedure. When you are finished deleting sessions, press the Close button. Note that you are not allowed to delete the session that is opened at the moment.



#### ***Current session***

You can see the name of the session you are using at the moment in the title bar of the program, and also in the bottom-right corner of the screen, under the

caption "Session" on the status bar (see image below).



#### ***11.7.2.7 Settings***

It is possible to adapt many settings in the MTD. Basically, there are three places where you can change settings. The first one are the settings for the entire program, such as settings with respect to the database the Framework uses to store data in. Next there are monitor settings. These settings are specific to a session, you can adjust the way in which the events are collected from the hardware. Finally, some plug-ins may offer the possibility to modify settings for that particular plug-in.

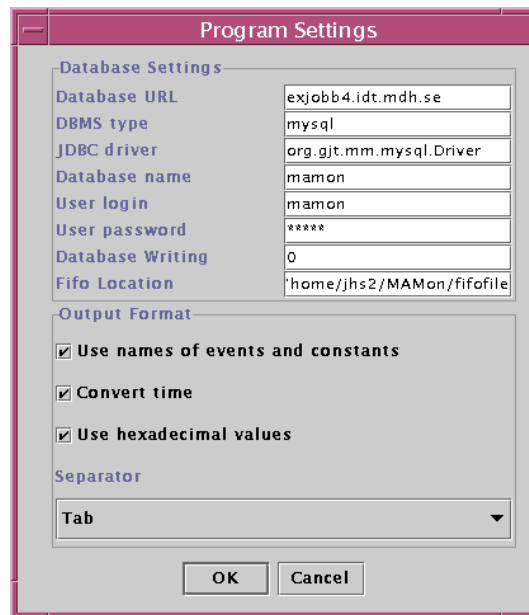
#### ***Program settings***

You can change the program settings using `Edit | Program Settings`. You will see a dialog divided in two sections (see image below). The first section deals with settings for the database, the second with the format in which events are printed within the program.

#### ***Database Settings***

The following items can be set regarding the database the Framework uses to store events and sessions:

- Database URL - the location of the database server
- DBMS type - the type of DataBase Management System\*
- JDBC driver - the driver needed to communicate with the database\*
- Database name - the name of the database that is used to store the data
- User login - the login name that should be used
- User password - the user password that should be used
- Database Writing - method used to insert event data in the database\*
- FIFO location - path to the FIFO the MTD uses when monitoring. This should be on the same machine as the database server.



The items marked by an asterisk should only be modified when you change the DMBS to use with the Framework. See Programmer's Guide for how to accomplish this.

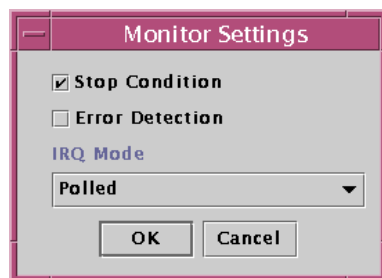
#### *Output Format*

The items can be set regarding the way in which events are printed throughout the program:

- Use names of events and constants - if defined, the MTD will use names for events and their contents instead of displaying plain numbers
- Convert time - the times at which the events occurred are displayed using the general time format, instead of a value in nanoseconds
- Use hexadecimal value - any numbers appearing are displayed in hexadecimal format instead of decimal
- Separator - this character is used to separate the different parts of an event

### **Monitor settings**

You can change the settings for the monitor by selecting `Edit | Monitor Settings`. You will now a dialog (see image below).



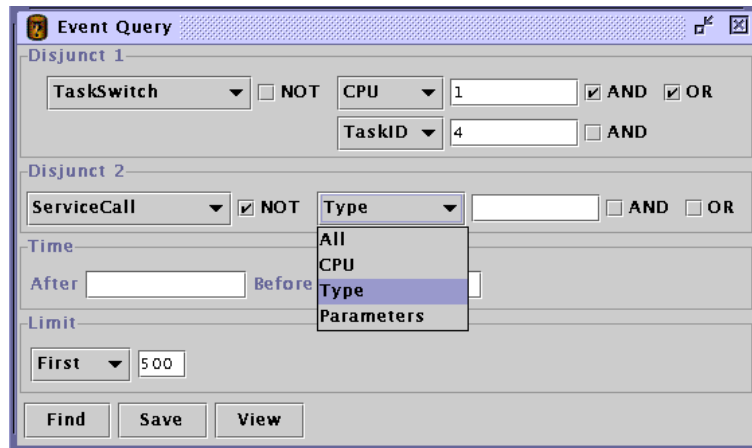
Here, you can set the following items:

- Stop Condition - if selected, the MTD will stop when the hardware buffer is full, otherwise, monitoring is continuous
- Error Detection - if selected, the MTD will attempt to verify whether the received data is correct by sending a confirmation.
- IRQ Mode - determines the way in which new events are detected. There are four options:
  - Polled
  - New event
  - Buffer half-full
  - Buffer full

### **11.7.3 Plug-in Tools**

#### **11.7.3.1 Event Query plug-in**

The Event Query plug-in allows you to make a selection of the events stored in the database.



### *Event type*

To restrict the type of events that is returned, select the event type from the first combobox in the Disjunct 1 section. If you want to search all types of events, select All here. In the above image, we search for TaskSwitch events in Disjunct 1.

### *Field requirements*

To find only events where specific fields have specific values, use the second combobox and the text box next to it. In the combobox, you can select the fields that are defined for the event type you selected. In the text box, you can then add the value you want the field to be. You can add a new field by clicking the AND box. In the above example, the events in Disjunct 1 have to be executed on CPU 1 and have TaskID 4.

To get events that do not have a certain value for a field, check the NOT box. If you do not want to enter a value for a field, select the field All.

### *Disjuncts*

If you want your query to return more than one type of event, click the OR checkbox. A new part will become visible, similar to the Disjunct 1 section, like shown in the above image. The results that you get will fulfill the requirements of the first disjunct or the second disjunct, or both. More disjuncts can be added clicking subsequent OR boxes.

***Time***

The events can also be restricted by time. If you fill in a value in the field After, only events that appeared after that time are displayed. For the field Before, only events before that time are displayed. You can also use both fields. The time values can be expressed in both nanoseconds and in the MTD's time format.

***Limit***

If you only want a limited number of events returned, instead of all events that match your requirements, use the Limit section. There are three type of limiting:

- First - you will get the first x events in time that match your query, where x is the number you type in the text box.
- Last - you will get the last x events in time that match your query, where x is the number you type in the text box.
- Range - you will get events x to y in time that match your query, where x and y are then numbers you type in the text boxes.

In the image shown above, the first 500 events that match the requirements will be returned.

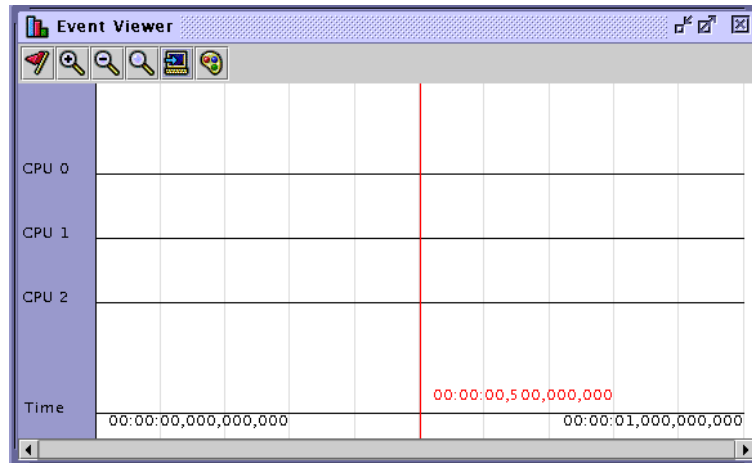
***Results***

The selected events can be displayed in three different ways:


- Find - this will output the events in text window
- Save - this will output the events to file. You can select the name of this file. The format is the same as used for the textual output with Find.
- View - this will display the time of the first event in a graphical plug-in. A new button will also appear, Next. This button shows the second event of the result.

***11.7.3.2 Event Viewer plug-in***

The Event Viewer presents a graphical representation of the events that occurred while monitoring. The events are displayed chronologically.



### ***Fetching data***

You can load data in the event view by pressing the button labeled with this icon: 


You will see a dialog in which you can enter the range in time that you want to display. **WARNING:** for long ranges, it may take some time to load and display. The times to enter here can be in nanoseconds or in the common time format. If you press OK, the events in the selected range will be loaded and displayed. The initial view will be of the entire range fetched, except in case this is longer than 1 minute; then the first minute will be visible.

### ***Panel***




After fetching, the panel displays a line for each CPU on the monitored system. Events taking place on these CPU are displayed along these lines. Tasks executing are shown as coloured bars, while other events are represented with icons. Events regarding the entire system are shown above the line of CPU 0. To get more details on a certain event, double-click it. A text-enabled plug-in will then show you the details of this event.

### ***Marker***


By clicking on the panel, you will see a red line drawn. This line indicates the time that point represents. In case you have lost view of the marker, press

the  button to find it again.

### **Zooming**

You can zoom in and out on the panel using the  and  buttons. The  button will bring you back to the original view. Exact zooming is also possible by dragging the mouse over the panel. The MTD will zoom in to the selected area, made visible by a red box, when the mouse button is released.

### **Settings**


You can set the colours that are used to for the tasks using the  button. You will see the following dialog:




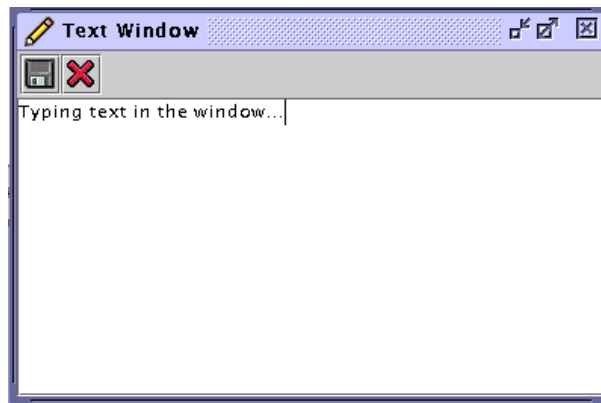
You can change the colour belonging to a task number by clicking the Set button next to the task number. For numbers higher than 15, use the lower text box. Enter the number and press enter to see the corresponding colour. You can set it using the set button next to the text box.



### 11.7.3.3 *Text plug-in*

The text plug-in consists of a text window, in which the program can display textual information. You can edit and modify the text, or add your own text. The contents of the window can be save on file by selecting Tools | Text Window | Save to File, or clicking .

To clear the contents of the text area, select Tools | Text Window | Clear or click .



## *Bibliography*

- [1] Mohammed El Shobaki and Lennart Lindh, *A Hardware and Software Monitor for High-Level System-on-Chip Verification*, In proceedings of the IEEE International Symposium on Quality Electronic Design, San Jose, CA, USA, March 2001.
- [2] Mohammed El Shobaki, *On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems*, In proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA), Tokyo, Japan, March 2002.
- [3] Mohammed El Shobaki, *HWMON – EPP/SPP IEEE1284 Host Interface*, Mälardalen Real-Time Research Center, Department of Computer Engineering, May 2001.
- [4] Andreas Malmquist, *MAMON 1.1 USB – Implementation Report*, Department of Computer Science and Engineering, April 2004.
- [5] Adil Al-Wandi, and Mehrdad Hessadi, *Monitoring of System Work Load*, M.Sc. thesis report, Department of Computer Science and Engineering, May 2002.
- [6] Johan Andersson and Toni Riutta, *DREA 2001 - The monitoring group*, Mälardalens Högskola, 2001.
- [7] Heijmans, Jeroen, *API Documentation MAMon*, Mälardalens Högskola, 2001.
- [8] Nunn, Gerald, *Conquer Swing deficiencies in MDI development*, <http://www.javaworld.com/javaworld/jw-05-2001/jw-0525-mdi.html>, 2001.
- [9] Sun Microsystems, Inc., *JavaHelp System User's Guide*, <http://java.sun.com/products/javahelp/index.html>.
- [10] Venners, Bill, *Design with dynamic extension - How dynamic extension works in Java and how to use it in your designs*, <http://www.javaworld.com/javaworld/jw-01-1999/jw-01-techniques.html>, 1999.
- [11] Heijmans, Jeroen, *MAMon - Project Report*, Mälardalens Högskola, 2001.

- [12] *JClass Chart Documentation*,  
<http://www.klg.com/software/support/jclass/docs/chart/>.
- [13] *JDBC Data Access API - Drivers*,  
<http://industry.java.sun.com/products/jdbc/drivers>.
- [14] *Java Standard Edition Platform Documentation*,  
<http://java.sun.com/docs/index.html>.
- [15] *Creating a GUI with JFC/Swing*,  
<http://java.sun.com/docs/books/tutorial/uiswing/index.html>.
- [16] *The Javadoc Tool Home Page*,  
<http://java.sun.com/j2se/javadoc/index.html>.
- [17] *A  $\LaTeX$  Generating Java 2.0 doclet*,  
<http://www.c2-tech.com/java/TexDoclet/>.
- [18] *Java Native Interface*,  
<http://java.sun.com/docs/books/tutorial/native1.1/index.html>.
- [19] Sun Microsystems, Inc., *JavaHelp System User's Guide*,  
<http://java.sun.com/products/javahelp/index.html>.



---

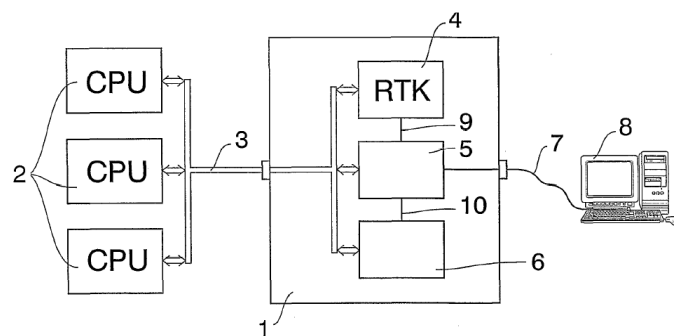
## Appendix A

# Patent

---

### *Abstract of WO02077810 / SE517917*

An integrated circuit (1) for a computer system, where the computer system comprises at least one processor (2) and an operating system which has at least one part (4) implemented in hardware, wherein said operating system part (4) is arranged in the integrated circuit (1). The integrated circuit (1) further comprises a unit (5) for supervising events in said computer system, wherein the supervising unit (5) has means for detecting events in the computer system and means for recording information about the events and that the supervising unit (5) is connected to said operating system part (4) in such a manner that information about internal events in the operation system part is possible to transfer to the supervising unit.





## (12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

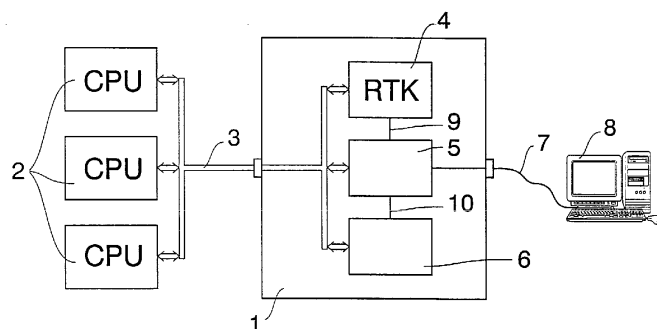
(19) World Intellectual Property Organization  
International Bureau(43) International Publication Date  
3 October 2002 (03.10.2002)

PCT

(10) International Publication Number  
WO 02/077810 A1

- (51) International Patent Classification<sup>7</sup>: G06F 9/46, 11/30
- (21) International Application Number: PCT/SE02/00561
- (22) International Filing Date: 22 March 2002 (22.03.2002)
- (25) Filing Language: Swedish
- (26) Publication Language: English
- (30) Priority Data:  
0101064-4 26 March 2001 (26.03.2001) SE
- (71) Applicant (for all designated States except US): RFO RE-ALFAST OPERATING SYSTEMS AB [SE/SE]; Sintervägen 14, S-721 30 Västerås (SE).
- (72) Inventor; and  
(75) Inventor/Applicant (for US only): EL SHOBAKI, Mohammed [SE/SE]; Hantverkargatan 11, S-722 12 Västerås (SE).
- (74) Agents: REYIER, Ann-Mari et al.; Bjerkéns Patentbyrå KB, Box 128, S-721 05 Västerås (SE).
- (81) Designated States (national): AE, AG, AL, AM, AT, AT (utility model), AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, CZ (utility model), DE, DE (utility model), DK, DK (utility model), DM, DZ, EC, EE, EE (utility model), ES, FI, FI (utility model), GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SK (utility model), SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZM, ZW.
- (84) Designated States (regional): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).
- Published:  
— with international search report
- For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: AN INTEGRATED CIRCUIT



(57) Abstract: An integrated circuit (1) for a computer system, where the computer system comprises at least one processor (2) and an operating system which has at least one part (4) implemented in hardware, wherein said operating system part (4) is arranged in the integrated circuit (1). The integrated circuit (1) further comprises a unit (5) for supervising events in said computer system, wherein the supervising unit (5) has means for detecting events in the computer system and means for recording information about the events and that the supervising unit (5) is connected to said operating system part (4) in such a manner that information about internal events in the operation system part is possible to transfer to the supervising unit.

SVERIGE

(12) **PATENTSKRIFT**(13) **C2**(11) **517 917**

(19) SE

(51) Internationell klass 7  
**G06F 11/30, 9/46****PATENT- OCH REGISTRERINGSVERKET**

(45) Patent meddelat **2002-08-06**  
 (41) Ansökan allmänt tillgänglig **2002-08-06**  
 (22) Patentansökan inkom **2001-03-26**  
 (24) Löpdag **2001-03-26**  
 (62) Stamansökans nummer  
 (86) Internationell ingivningsdag  
 (86) Ingivningsdag för ansökan om europeisk patent  
 (83) Deposition av mikroorganism

(21) Patentansökningsnummer **0101064-4**

Ansökan inkommen som:

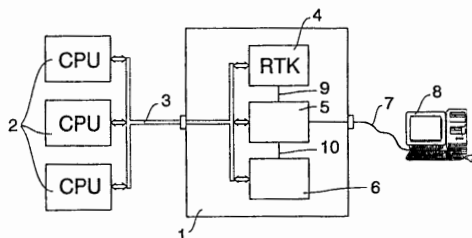
- svensk patentansökan fullföljd internationell patentansökan med nummer  
 omvändad europeisk patentansökan med nummer

(30) Prioritetsuppgifter  
- -(73) PATENTHAVARE **RFO RealFast Operating Systems AB, Sintervägen 14  
721 30 Västerås SE**(72) UPPFINNARE **Mohammed El Shobaki, Västerås SE**(74) OMBUD **Bjerkens Patentbyrå KB**(54) BENÄMNING **Integrerad krets för övervakning av händelser i datorsystem**

(56) ANFÖRDA PUBLIKATIONER: - - -

(57) SAMMANDRAG:

En integrerad krets (1) för ett datorsystem, där datorsystemet innefattar åtminstone en processor (2) och ett operativsystem, vilket åtminstone har en del (4) som är implementerad i hårdvara, varvid nämnda operativsystemdel (4) är anordnad på den integrerade kretsen (1). Den integrerade kretsen (1) innefattar vidare en enhet (5) för att övervaka händelser i nämnda datorsystem, varvid övervakningsenheten (5) har medel för att detektera händelser i datorsystemet och medel för att spela in information om händelserna, och att övervakningsenheten (5) är ansluten till nämnda operativsystemdel (4) på så sätt att information om interna händelser i operativsystemdelen är överföringsbara till övervakningsenheten.





**Description of WO02077810  
Translated from SE517917**

## **An Integrated Circuit**

### *A.1 Field of the invention*

The present invention refers to an integrated circuit for a computer system comprising at least one processor and an operating system, which at least has a part that is implemented in hardware, wherein said operating system part is arranged on the integrated circuit.

### *A.2 Prior art*

In a computer system, it is desirable to be able to detect and record different events in the computer system for the purpose of analyzing, measuring the performance, and testing the computer system. This is particularly important during the development of a new system. From now on, supervising of a computer system relates to the collection of information regarding the behavior of the computer system. The supervision may also extract or validate design parameters, such as execution times and delays in the system.

It is known to supervise what happens in a computer system through specific program instructions being located in the software code of the system. When those instructions are executed, events are initiated, and information about the generated events is stored for later analysis. A drawback with handling the supervision by software instructions in the program code is that the software of the system becomes overloaded with those extra instructions and that they might influence the timing in the system. The introduced disturbances caused by the extra instructions might also change the behavior of the program and when the software test is completed and the extra instructions are removed new errors might occur due to the change of the timing.

It is particularly important in a real-time operating system that the supervision does not influence the timing. Real-time systems are different from most other systems, since they are tremendously sensitive to disturbances in the timing. Typical for a real-time operating system is that it is deterministic, which means that it shall always be predictable. Thus, the supervision of a real-time system is not allowed to change the timing or the order of events in the system.

Another known method for supervising a computer system is the use of a supervisory device implemented in hardware. Such a supervisory device com-

prises probes, which are connected to buses and signals in the computer system. The supervisory device detects passively what is going on at the bus and collects information about events, which might be of interest. The advantage with hardware supervision is that it does not interfere with the system that it supervises. A problem in connection with the use of a separate supervisory device is that, in certain modern computer systems, many functions are physically integrated in the same circuit in the same enclosure. Therefore, it may be impossible to reach all signals necessary to achieve a good supervision. The enclosure of an integrated circuit should preferably not be too large and thus it has a limited number of pins. To obtain all signals necessary for the supervision, the number of pins must be considerably increased and thus the size of the enclosure must be increased and that is not desirable.

For the purpose of enhancing the performance in a computer system, it is known to implement the whole or at least a part of the operating system in hardware in an external unit outside the processor. High performance is particularly important in applications performed in real-time, such as, for example in process control applications. A hardware operating system has a realtime kernel arranged on an integrated circuit that is physically separated from the processor and the real-time kernel is communicating with the processor via a bus connection. It is not suitable to supervise such a computer system with software, since the timing in the system might be influenced. It is true that a supervisory device can be connected to the bus between the processor and the hardware operating system, but the information accessible on the bus is not enough to perform a reliable supervision of the system or to detect errors in the program code.

### *A.3 Summary of the invention*

The object of the inventions is to provide an integrated circuit comprising a hardware operating system or at least a part of an operating system which is implemented in hardware and which makes it possible to achieve a reliable supervision of events in the computer system.

This object is achieved with the initially described integrated circuit, which is characterized in that it further comprises a unit for supervising events in said computer system, wherein the supervising unit has means for detecting events in the computer system and means for recording information about the events, and that the supervising unit is connected to said operating system part in such a manner that information about internal events in the operating system part is possible to transfer to the supervising unit. By arranging a supervising

unit in the same integrated circuit as a hardware operating system part and connecting them to each other, necessary information for the supervision, for example information about internal events in the operating system part, can be transferred to the supervising unit.

Such internal information is not obtainable on the bus between the processor and the operating system part. The information about internal events is usually inaccessible or so sensitive to disturbances, such as capacitances in the cables, that it is not measurable. The supervising unit is passive and listens to what is going on in the operating system without influencing the system and its timing. When the supervising unit has detected an interesting event, it records the information about the event.

This information may later be analyzed, for instance for the purpose of detecting errors. Since the supervising unit is implemented in hardware, nothing prevents it from being used for supervision of arbitrary physical signals in the system.

In a preferred embodiment of the invention, said operating system is a real-time operating system and said operating system part is a real-time kernel. The invention is particularly advantageous if the operating system is a real-time operating system, since the timing in the system is not influenced by the supervision according to the invention.

In an embodiment of the invention, the operating system part comprises means for handling when a number of tasks shall be executed by the processor and said information comprises information about the current state of the tasks. This is important information and knowledge about it contributes to an improved supervision of the system. This information is difficult to obtain outside the integrated circuit, but the fact that the supervising unit and the hardware operating system part are arranged on the same integrated circuit makes it possible to transfer this information to the supervising unit in a simple way.

In an embodiment of the invention, the operating system part is communicating with one or a plurality of processors via a bus and the supervising unit comprises means for receiving information from the processor/processors from said bus. Thanks to the fact that the supervising unit is connected to the bus, a processor can address the supervising unit and transfer information about what is going on in the processor to the supervising unit.

The software is provided with program instructions for transferring information via the bus to the supervising unit. Thus, the supervision of the software can be based on events in the operating system part and on arbitrary information generated by the software itself. Accordingly, a combination of hardware and software supervision is obtained. This means that the supervision is further

improved, since the information about internal events in the operating system is supplemented with information about events in the software. Since most of the supervision is done in the hardware, only a few extra instructions in the software code are necessary and thus the load on the system is not so large as when all supervision is done in the software. Since the load on the system becomes low, the extra instructions do not necessarily have to be taken away when the supervision is finished and problems with the timing is thus avoided.

All information about events in the processor is not available on the bus. In an embodiment of the invention, the integrated circuit comprises at least one processor, which is connected to the supervising unit in such a manner that information about internal events in the processor is possible to transfer to the supervising unit. By integrating one or a plurality of processors in the computer system in the same circuit as the operative system part and the supervising unit, it is possible to transfer information about internal events in the processor to the supervising unit.

Thus, further improvement of the supervision is achieved.

In an embodiment of the invention, the integrated circuit comprises means for transferring the stored information to an external unit for further analysis of the information. The external unit may, for example, be a computer. The recorded and stored events are sent through a communication link to the external unit where they are stored in a database. The collected information may later, for example, be used for event based debugging, analysis of the behavior, and for statistics.

#### *A.4 Description of the drawings*

The present invention will now be explained by the description of different embodiment and with reference to the appended drawings (Section A.7).

Fig. 1 shows a block diagram of a computer system with an integrated circuit according to an embodiment of the invention.

Fig. 2 shows a block diagram of a supervising unit arranged on the integrated circuit in Figure 1.

Fig. 3 shows the format of an event package.

Fig. 4 shows another embodiment of an integrated circuit according to the invention.

## A.5 Description of embodiments

A computer program is structured in different tasks. A task is a sequentially arranged program and performs a predetermined function. In a real-time operating system, the tasks are given mutually priority and if two tasks are to be executed at the same time by the processor, the task with the highest priority is executed first. The means handling when a certain task is to be executed by the processor is denoted a scheduler and forms the kernel in a real-time operating system.

Figure 1 shows a computer system comprising an integrated circuit 1 according to the invention and three processors 2 arranged in parallel. The processors 2 and the integrated 1 circuit are connected to each other via a system bus. The computer system has a real-time operating system comprising a real-time kernel 4, which is implemented in hardware (RTK = Real Time Kernel) in the integrated circuit 1. The real-time kernel 4 is arranged in such a manner that it mainly executes the same functions as traditional real-time operating systems which are implemented in software do, for example handling, scheduling, and giving priority to tasks, communication, and synchronization between the tasks as well as interruption handling. More about how a real-time operating system is constructed is described in the book "Utilization of Hardware Parallelism in Realizing Real TimeKernels" by Lennart Lind, ISBN 0280-4506.

The integrated circuit further comprises a supervising unit 5 connected to the real-time kernel 4. The function of the supervising unit is to detect and record events in the computer system. Events in the operating system is, for example : – a system call to the operating system, such as create task, delete task, send message, receive message, – the state of the task is changed, such as setting a task in a blocked state or unlock it from the blocking, and – interruption request from external units.

For the purpose of supervising events in the operation system, the supervising unit 5 is connected directly in the hardware to a number of selected signals in the real-time kernel 4. The connection is implemented as one or a plurality of physical wires 9 from internal signals in the real-time kernel 4 to the supervising unit 5. In such a way, it is possible to get access to valuable information about what is happening in the operating system.

Those internal signals comprise i. a. information about the condition of the tasks in the system, the priority of the tasks, internal communication activities as well as internal and external interruptions.

In certain cases, the ability to generate events from the software is necessary, for example to see whether certain control points are passed or to report

the contents in a certain memory. Such events can be produced by introduction of software instructions in the program code. The supervising unit can also be used for recording general information, directly generated from the software, i. e. information coming directly from the execution of the software on one or a plurality of processors 2 in the system. For the purpose of supervising events in the software and to receive information from the software, a supervising unit 5 is connected to the system bus 3. The software generates information through the system bus by writing data to a particular register in the supervising unit 5.

It is also possible to connect the supervising unit to an arbitrary signal or hardware unit in the integrated circuit for supervising the signal or the hardware unit. Such hardware units are buses and internal memories. This means that the supervising unit may also function as a general logic analyzer. Thus, the supervising unit 5 can be used to detect functional errors in the hardware.

This is particularly useful in situations where conventional error localization methods are difficult to use, for example due to high system speeds or the fact that the signals are difficult to reach because of a limited number of pins in the integrated circuit. A hardware unit 6 in the integrated circuit 1 is connected through physical wires 10 to the supervising unit 5.

The supervising unit 5 is further connected through a parallel communication link 7 to an external computer 8. The supervising unit 5 listens passively to events, logical and/or on system level, in the real-time kernel 4 and interesting events are recorded.

The recorded events are then transferred to a database in the external computer where further handling and analysis of the information occur.

The integrated circuit can either be implemented in ASIC (Application Specific Integrated Circuit) or in a programmable hardware such as FPGA (Field Programmable Gate Array). All the components are integrated on the same plate, which preferably is made of silicon. Figure 2 shows a block scheme of the internal construction of the supervising unit 1. The supervising unit comprises an event detector 11, a timer 12, an event recorder 13, an event buffer 14, and an interface 15 to the external computer. Input signals to the supervising unit are hardwired signals 9 from the real-time kernel, signals on the system bus 3 and signals 10 from the hardware unit 6. The input signals are received by the event detector 11 that detects whether any event to be recorded has occurred. The event detector 11 is a comparator that compares the input signals with internal predetermined conditions. There are a number of conditions defining which events to be supervised and those conditions are hard coded in the event detector.

The event detector detects whether an event has occurred by comparing the

input signals with the predetermined conditions.

The event detector 11 comprises three different detectors 16-18 with different predetermined conditions. The first detector 16 receives the internal signals 9 from the real-time kernel and detects events in the operating system. The second detector 17 comprises a register receiving information from the software via the bus 3 and detects when information is written to the register.

The third detector 18 receives the signals 10 from the other hardware units and detects events in the hardware units.

When any of the conditions is met, i. e. an event is detected; the event is reported further to the event recorder 13. The object of the event recorder is to produce a package with information about the event, which package later can be transferred through the interface 15 to the external computer. The event recorder 11 is activated by one or a plurality of start signals from the event detector and receives at the same time an identification number from the event detector. The identification number presents information to the event recorder about which signals to be packed together with the detected event. When the event recorder is activated, it creates an information package comprising the identification number of the event, the signals connected to the event, and the time when the event was detected. The event package must have a certain predetermined format. The time is read from the timer 12 that is connected to the event recorder 13.

The event package is then stored in the internal event buffer 14.

This buffer is organized as a FIFO queue, i. e. the oldest package comes out first from the queue and the newest package comes out last (FIFO = First In First Out). At the same time as the event packages are stored in the event buffer, an indicator of a detected event is stored in a status register (not shown).

The interface 15 transfers the packages to the external computer. Through the interface, the external computer may have an indication of the fact that an event has occurred and reads the event package from the event buffer 14. Indication about whether an event has been detected or not can either be obtained by reading the information in the status register or by programming the interface, so that it automatically generates a signal when new events are available in the event buffer. The size of the event buffer can be varied and depends on the application.

Sometimes one does not wish to transfer all detected events to the external computer, for example, if the number of events is large. In the event detector 11, there is a filter 19 implemented as a programmable register. The filter 19 makes it possible to choose interesting events and only letting those through. The filter 19 is connected to the interface 15 and it is possible to send in-

structions from the external computer to the filter about which events to be let through the filter.

Figure 3 shows an example of the design of a package having information about an event. The event package comprises three different fields of information, a first field 20 comprising the identification number of the event, i. e. information about the type of event, a second field 21 comprising the time, and a third field 22 comprising more information about the event in the form of a plurality of parameters. An example of an event is when the processor begins and ends execution of a new task. Parameters in the third field should then comprise information about the identification number of the new task to be executed and which one of the processors to be executing the task. For the event "send message", the parameters should comprise identification number for the receiving task and a pointer to the message.

Figure 4 shows an embodiment of the invention, wherein an integrated circuit 29 comprising a processor 30, a real-time kernel 4 of the operating system, a supervising unit 5, and various other types of hardware 6. The computer system in this embodiment has only one processor and it is integrated in the same circuit as the real-time kernel 4 and the supervising unit 5.

For a computer system comprising a plurality of processors, it is possible to integrate all the processors in the same circuit. The supervision unit 5 is hardwired to a number of internal signals in the processor 30. Thus, the supervising unit will have access to internal information about the events not visible on the system bus. Examples of such signals are signals to and from Cache memories in the processor. In the same way as in the previous embodiment, the supervising unit 5 is connected to internal signals in the real-time kernel 4 and to other hardware functions 6 in the circuit.

The invention is not limited to the disclosed embodiments but may be varied and modified within the scope of the following claims.



## A.6 Claims

1. An integrated circuit (1) for a computer system, where the computer system comprises at least one processor (2, 30) and an operating system which has at least one part (4) implemented in hardware, wherein said operating system part (4) is arranged in the integrated circuit (1), characterized in that the integrated circuit (1) further comprises a unit (5) for supervising events in said computer system, wherein the supervising unit (5) has means (11) for detecting events in the computer system and means (13) for recording information about the events and that the supervising unit (5) is connected to said operating system part (4) in such a manner that information about internal events in the operation system part is possible to transfer to the supervising unit.
2. An integrated circuit according to claim 1, characterized in that said operating system (4) is a real-time operating system and that said operating system part is a real-time kernel.
3. An integrated circuit according to claim 1 or 2, characterized in that the operating system part (4) comprises means for handling when a number of tasks shall be executed by the processor and that said information comprises information about the current state of the tasks.
4. An integrated circuit according to any of the claims 1–3, characterized in that said operating system part (4) is communicating with at least one processor (2, 30) through a bus (3) and that the supervising unit (5) comprises means (17) for receiving information about the events in the processor from the bus.
5. An integrated circuit according to any of the previous claims, characterized in that it comprises at least one processor (30), which is connected to the supervising unit (5) in such a manner that information about internal events in the processor is possible to transfer to the supervising unit.
6. An integrated circuit according to any of the previous claims, characterized in that it comprises means (15) for transferring the recorded information to an external unit (8) for further analysis of the information.
7. An integrated circuit according to any of the previous claims, characterized in that it is composed of a system integrated on silicon.

A.7 Drawings

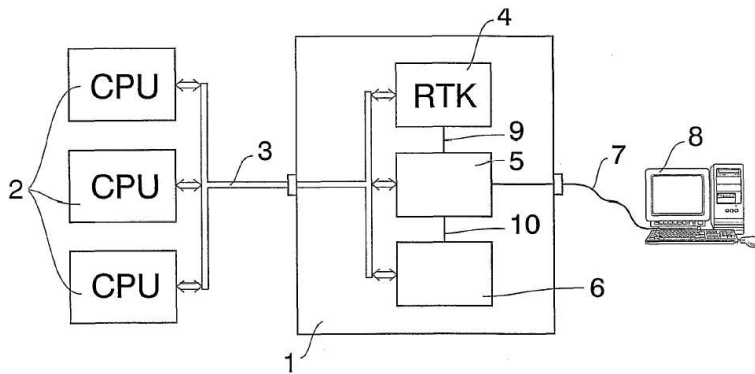


Fig. 1

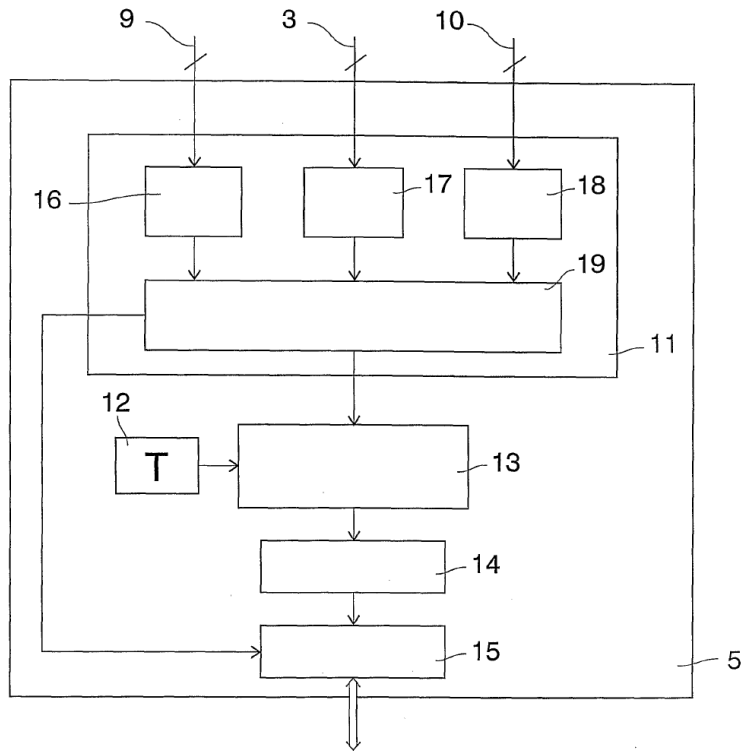


Fig. 2

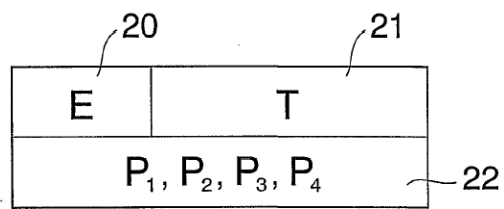


Fig. 3

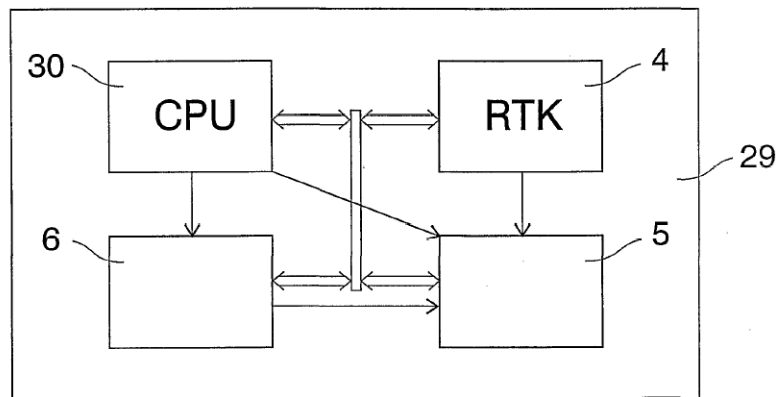


Fig. 4





## Recent licentiate theses from the Department of Information Technology

- 2003-006** Tomas Olsson: *Bootstrapping and Decentralizing Recommender Systems*
- 2003-007** Maria Karlsson: *Market Based Programming and Resource Allocation*
- 2003-008** Zoran Radovic: *Efficient Synchronization and Coherence for Nonuniform Communication Architectures*
- 2003-009** Martin Karlsson: *Cache Memory Design Trade-offs for Current and Emerging Workloads*
- 2003-010** Dan Wallin: *Exploiting Data Locality in Adaptive Architectures*
- 2003-011** Tobias Amnell: *Code Synthesis for Timed Automata*
- 2003-012** Olivier Amoignon: *Adjoint-Based Aerodynamic Shape Optimization*
- 2003-013** Stina Nylander: *The Ubiquitous Interactor - Mobile Services with Multiple User Interfaces*
- 2003-014** Kajsa Ljungberg: *Numerical Methods for Mapping of Multiple QTL*
- 2003-015** Erik Berg: *Methods for Run Time Analysis of Data Locality*
- 2004-001** Niclas Sandgren: *Parametric Methods for Frequency-Selective MR Spectroscopy*
- 2004-002** Markus Nordén: *Parallel PDE Solvers on cc-NUMA Systems*
- 2004-003** Yngve Selén: *Model Selection*
- 2004-004** Mohammed El Shobaki: *On-Chip Monitoring for Non-Intrusive Hardware/-Software Observability*



UPPSALA  
UNIVERSITET