SysMon – A framework for monitoring and measuring real-time properties

Master Thesis, Computer Science

Spring 2012

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

Authors:
Fredrik Nilsson (fnn05003@student.mdh.se)
Andreas Pettersson (apn07010@student.mdh.se)

Supervisor:
Mikael Sjödin (mikael.sjodin@mdh.se)
Examiner:
Moris Behnam (moris.behnam@mdh.se)

Abstract

ABB SA Products designs and manufactures complex real-time systems. The real-time properties of the system are hard to measure and test especially in the long run, e.g. monitoring a system for months out in the real environment. ABB have started developing their own tool called JobMon for monitoring timing requirements, but they needed to measure more properties than time and in a more dynamic way than JobMon is constructed today. The tool must be able to measure different kind of data and be able to be monitor as long as the system itself.

This thesis first does a survey and evaluation on existing commercial tools and if there exists a tool that can be integrated to the system and fulfill all demands. Different trace recorders and system monitoring tools are presented with its properties and functions. The conclusion is that there is no such tool and the best solution is to design and develop a new tool.

The result is SysMon, a dynamic generic framework for measuring any type of data within a real-time system. The main focus for measuring during this thesis is time measurements, but no limits or assumptions of data types are made, and during late steps of the development new types of measurements are integrated. SysMon can also handle limits for measurements and, if required, take pre-defined actions e.g. triggering a logging function and saving all information about the measurement that passed the limit.

The new tool is integrated to the system and evaluated thoroughly. It is an important factor to not steal too much resource from the system itself, and therefore a measurement of the tool’s intrusiveness is evaluated.

Sammanfattning

ABB SA Products designar och konstruerar komplexa realtidssystem. Realtidsegenskaperna för systemen är svåra att mäta och testa, speciellt under långa tidsperioder, t.ex. under drift i dess riktiga miljö under månader av online tid. ABB SA Products har börjat utvecklat ett eget verktyg, JobMon, för att kunna övervaka och mäta egenskaper i form av tid. Men behovet är större än att endast mäta tid och alla möjliga slags data behöver övervakas och utvärderas.

Det här examensarbetet gör först en undersökning och utvärdering av existerande kommersiella verktyg och om det redan finns ett verktyg som uppfyller alla krav. Olika tracerecorders och systemövervakningsverktyg är presenterade med dess egenskaper och funktioner. Slutsatsen är till sist att det inte finns något existerande verktyg och att den bästa lösningen är att utveckla ett nytt verktyg.

Resultatet är SysMon, ett dynamisk generisk ramverk för att mäta vilken form av data som helst. Huvudfokus under examensarbetet är tidsmätningar, men inga antaganden om vilka datatyper som kan användas görs. Under den senare delen av examensarbetet implementeras också en ny typ av mätning i system ticks. SysMon kan också hantera gränser för mätningar och, om nödvändigt, exekvera fördefinierade funktioner, t.ex. trigga en loggning och spara nödvändig information om mätningen som överskred gränsen.

Det nya verktyget blir integrerat i systemet och testat noggrant. Det är viktigt att verktyget inte tar för mycket resurser från det normala systemet och därför utförs även en utvärdering av hur resurskrävande verktyget är.

Acknowledgements

We would like to thank our advisor Leif Enblom for support and time effort during this thesis. We would also like to thank Arve Sollie for his input and suggestions, Lasse Kinnunen and all other developers for answering our questions about the system.

A big thanks also goes to Henrik Johansson for letting us do the thesis at the department and also MDH advisor Mikael Sjödin for all input and comments.

Abbreviations and terms

BCET	Best Case Execution Time
EDF	Earliest Deadline First
FPS	Fixed Priority Scheduling
I/O	Input/output
JobMon	Job Monitor tool – the tool developed earlier at ABB
OID	Object Identifier
PCP	Priority Ceiling protocol
PIP	Priority Inheritance Protocol
RTOS	Real-time Operating System
SysMon	System Monitor tool – the tool developed during this thesis
WCET	Worst Case Execution Time

List of figures

List of Tables

Table of Contents

1 Introduction

Embedded computers are getting more and more common. Today they are the most common type of computers manufactured. Many of these serve important functions in the human society, e.g. a car often have tens of embedded computers to control all functions. The usage areas of embedded systems are almost ubiquitous and there are still several areas that have not taken the step from analog electronics to digital microprocessors.

Many of the embedded systems are time critical and are often referred to as real-time systems. These systems have specific requirements with time aspects.

A lot of these systems have hard timing requirements and a system that executes too fast or too slow will result in a bad, or even dangerous, system. A good and easy understandable example of a time critical system is the airbag inflation in a car. It is important that it gets inflated exactly at the right time and not too early or too late.

The problem is that it is not always an easy matter to monitor and measure large complex real-time system in respect to their timing behavior when the system consists of a large amount of tasks and threads. The systems have also often been developed by several persons during tens of years, which often make it hard for one person to have a complete understanding of the whole system.

This thesis looks into the possibilities for monitoring a large industrial real-time system and gives a suggestion of a solution to the analysis problem.

1.1 Purpose

Currently there are different types of analysis tools in use at ABB SA Products, referenced to as ABB in the report, scaling from the highest to the lowest level. In the highest end there is simple CPU usage and the next level is CPU usage division between system tasks. On the lowest, most specific, level there is e.g. System Viewer, which is Wind River's official debug utility for VxWorks.

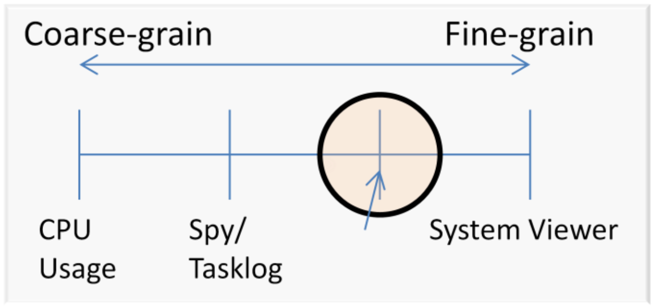


Figure 1: Grouping of analysis tools

The problem is that there is a gap between simple CPU usage surveillance and System Viewer, shown in Figure 1. What is needed is a program that can be used for long term monitoring of a system execution without the need of human interaction. A tool called JobMon has been developed by ABB and is in a research state. The tool gives the user possibility to detect errors like deadline misses and jitter of task execution. It has a static implementation and today only allows for five time measurements. It does not include any alarm functionality and requires that a person is continuously takes manual snapshots of the tool output. Even if you can see that errors have occurred, it is impossible to know exactly when it happened, since it can be whenever between the manual snapshots.

The purpose of the thesis is to look into the possibility to either utilize and integrate an existing tool or to develop a more advanced version of the existing company developed tool JobMon. The tool should be used as a long-term monitoring tool that can run in the background of a system test and warn when pre-defined errors in execution have been found. With the help of this tool, important system properties can be monitored and pre-defined error states would be possible to automatically detect. This error detecting tool should also be able to write logs over the system execution history or interact with a third party trace log writer.

1.2 Case-study description

The techniques proposed in this thesis will be demonstrated in a case-study using a protective relay developed at ABB.

Protective relay are used to protect the power transmission systems. The core idea is the same as normal household fuses, to protect and maintain as large part of the systems operational as possible upon failure.

Electricity can be transferred for many miles, and it is not unusual that something affects the power lines, e.g. trees falling over or hit by lightning. If not treated correctly this might affect the end customer and/or the infrastructure of the power lines in a negative way.

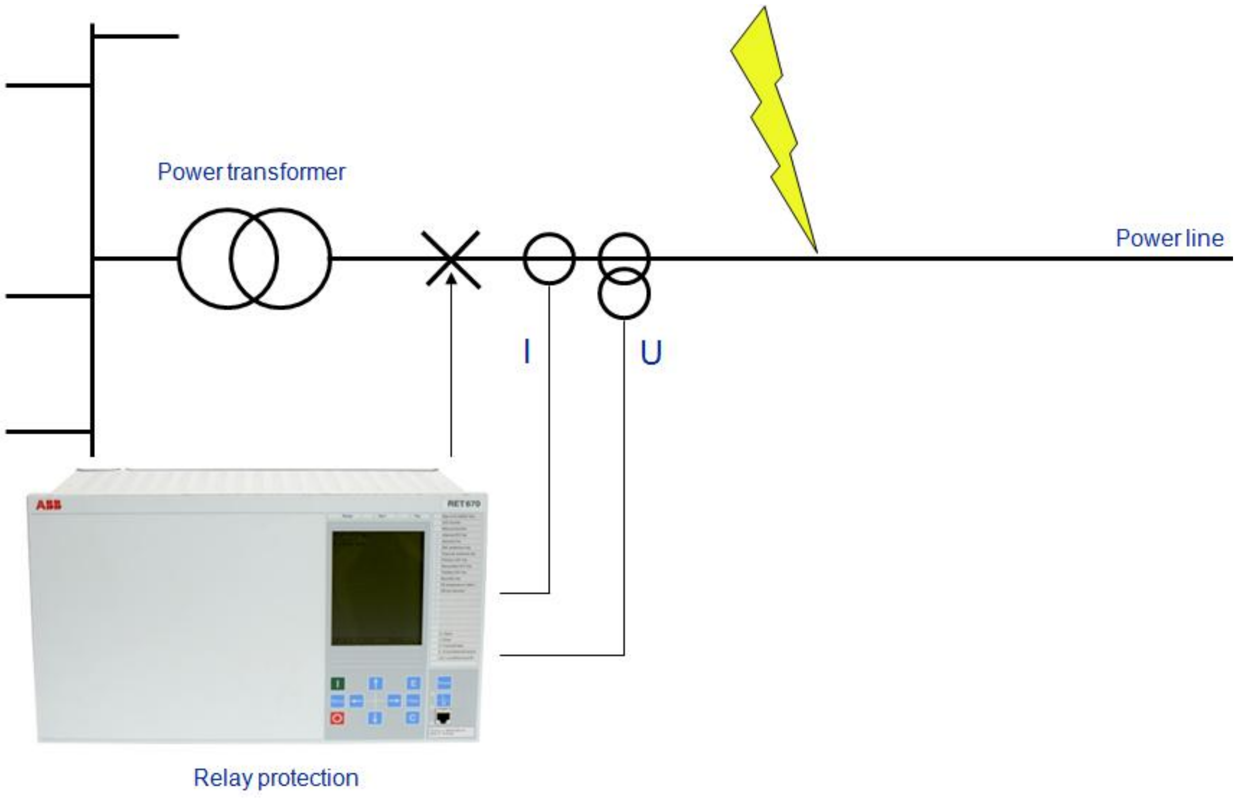


Figure 2: Protective relay (ABB internal picture)

Protective relays have secured our power lines since 1903 when ASEA developed the first mechanical relay [1]. Over the years the complexity and functions have increased and today they are digital intelligent embedded computers. Multiple units can be linked together to increase the ability to detect failures and managers of the systems can monitor and set important parameters far from the physical position.

The protective relays must trip the circuit breaker when it detects a possible failure on the power line. Detection of failures is e.g. done by measuring the current on two nearby places and calculating if they differ or by a simple voltage meter. Since the protective relay only protects a smaller part of the total power system it will only take the smaller subsystem out of order. This will maintain the functionality in all other parts. The intelligent relay can also, upon failure, notify a predefined technician by e-mail or SMS. When the technician has been alerted he or she can connect to the relay to gather information of why and where it occurred [1].

Due to the nature of electricity the circuit breaker trip has to be done quickly to avoid damage or potential danger for the end customer. This is one function where the real-time system plays an important role. Since there are a lot of things going on in the system, e.g. communication and measurements it is important to keep track of the system behavior at all times to guarantee the e.g. circuit breaker trip timing functionality and communication link timeouts.

1.3 Problem formulation

The problem that the company wants to solve is a gap in the types of system analysis tools that they currently have.

Figure 1 shows a scale of analysis tools stretching from the most basic type to the most advance type of analysis. The basic tools measures only pure CPU usage and just shows the amount of CPU used for a defined time frame. This gives the user an idea of the total load of the system but no information on what is using the system.

The next step is logging of single tasks and their CPU usage. This could be interesting to spot a task that is using a lot of CPU time but still doesn't tell the user about the actual system execution. On the right end of the scale there are analysing tools like System Viewer.

The information gathered from the right side often has a lot of details. These tools also use a lot more resources and are intrusive. Usage of system resources while monitoring can have unknown amount of side effect, and the monitored system might not act the same without the monitoring tool. The tools of the left side are using less resource but also providing less information.

Information intensive tools like SystemViewer are often used when you know that you have a problem and you also know where it is in the execution trace. This makes it possible to log a few seconds by streaming it in real-time to a PC or writing a log file for offline analyzing. The log can then by analyzed by using the graphical tool of SystemViewer and to, hopefully, find the root cause of the situation.

Instead consider a situation where there is an error that shows once every month, it would be impossible to use this kind of logging. The logs would be huge and finding the root cause would probably be like finding a needle in a haystack.

What the company wants is to fill the gap between analysing tools like System Viewer and Task log. The monitoring tool should be able to guarantee that, during the products uptime, nothing bad have happened. It should be active at all times and monitor for system failures. The tool should have the possibility to record information on the system continuously in the background and stop recording at user defined events (like a deadline miss or a buffer overflow). The log file must then contain enough information for a manual offline analyze of what went wrong.

ABB have also stated that they want to measure and/or evaluate a lot of different parameters. One type of measurement, e.g. time between two events, will not be enough. The framework for evaluating and measuring the system needs to be easy to extend with custom designed probes and a custom designed evaluator to decide whether the probe(s) have good or bad value.

2 Background

The background section gives the reader a theoretical base for the rest of the thesis work. It is used for introducing important factors of a real-time system and properties to take into consideration when designing a monitoring tool. The section gives both general information regarding real-time systems and more specific information about monitoring, measuring and analyzing real-time system behaviors.

2.1 Real-time systems

A real-time system has much in common with regular computer system but with one big difference. In a real-time system it is not only a correct execution that defines that the system is working but also the time frame in which the task is done.

When working with a real-time system it is important to have timing guarantees so that all tasks are done exactly when they are supposed to. If critical tasks are executed with a jitter of just milliseconds, the system could be performing so bad that it might be considered useless or in worst case even dangerous.

To understand how problems in these systems can occur, some basic functionality, properties and common issues are explained.

2.1.1 Hard versus soft systems

Real-time systems are divided into two different types. These are systems with soft timing requirements and hard timing requirements.

In hard real-time systems the timing is of main importance and missed deadlines and jitter is considered a malfunctioned system. A classic example of a hard real-time system is a car airbag. It is not enough that the airbag is inflated sometime after a collision; it has to be inflated at exactly the right moment. If it is inflated to early or too late, it will not help, or even do more damage than not inflated at all.

In soft real-time systems the time demands are a bit less. If deadlines are missed the system is considered bad, but it is not as critical. One example of this is a DVD-player. If the task that handles a video stream misses a deadline it might be a glitch in the video playback. This is irritating for the user but the DVD-player will still continue to work.

2.1.2 Event-triggered versus time-triggered system

There are two main types of systems; event-triggered and time-triggered [2].

Event-triggered systems are based on that the system receives different events that starts job in the system [3]. An event can e.g. be an I/O that triggers an interrupt routine. Since the scheduling becomes dynamic it is impossible to determine the maximum execution time without taking into account synchronization and interactions with other tasks. The events are often happening in a non-deterministic way and it is therefore impossible to calculate the peak load performance [4].

Testing the system is the only way to get a good estimation of its behavior and high load performance. Since events happen randomly it is also often a must to not just test it in the real environment, but also in a sort of worst case simulated environment. This is because events that produce the peak loads often happen rarely in the real environment [4] and it is mostly the extreme situations of the systems that are most important and most interesting.

It is also important to determine if the test patterns, used for pushing the system to extreme states, are something that actually is possible to happen in the real environment for the system [4].

When an event happens the system is often supposed to give some kind of response back. The worst case execution time from an event to response is an important property of an event triggered system.

Time-triggered system is based on a clock which triggers interrupts. These interrupts are the only ones the system will receive and determines release times for tasks [5]. When a task is released it is placed in the ready queue and the scheduling algorithm of the system will determine when the task gets to execute.

It is easier to calculate maximum execution time for tasks for a time triggered system than for an event-triggered. This is due to that one can predict how the tasks will interact and synchronize beforehand [4]. This makes time triggered systems predictable [6] since it will, at all times, execute according to the preconfigured schedule.

Scheduling of the tasks is done offline and it is possible to lookup which task to execute on a clock interrupt, according to the predefined schedule placed in a table or similar. Time-triggered scheduling is often also called static scheduling [6].

2.2 Tasks and priorities

As systems grow larger it gets more and more complex. When more code is added with different work areas, it is a good idea to separate these codes to different tasks that runs in separate threads in the system. These different threads can have different time constraints and importance to the system. Often there is an outside stimulus to respond to and not only correct answer is needed, but also within correct time interval [3]. Because of this it is necessary to be able to design the system with different priorities and scheduling so that important threads have the chance to execute in time.

Priorities are assigned to tasks before system execution. When tasks are ready to execute, the system uses the priorities to decide on which task that gets to execute. In what way this decision is taken and what criteria's that is taken into consideration is explained in the sections after this.

The assignment of priorities to tasks in a system is not an easy matter and there have been much research on different algorithms for assigning priorities. Two good strategies commonly used this are Rate monotonic and deadline-monotonic.

Rate monotonic uses the period times of tasks to decide the priorities. The highest priority task is the one with the shortest period time, and vice versa. The rate-monotonic algorithm is only used in systems where tasks have the same deadline as their period time [7]. An extension to rate monotonic is the deadline-monotonic algorithm. This algorithm uses the task deadlines as the base for priorities; the task with shortest deadline has the highest priority in the system [8]. This means that the algorithm can be used in systems where tasks have different period times and deadlines.

Rate monotonic and deadline monotonic is mostly good to use in smaller system. For more complex system it could be appropriate to use a more advanced priority assignment algorithm. One example of this is Audslyes algorithm [9]. This algorithm presents a way of assigning priorities in system where tasks have arbitrary release times, which means that there are no point in the system where all tasks are released simultaneously. With the help of the algorithm, systems where tasks have different release times can be scheduled at scenarios where rate-monotonic and deadline-monotonic priority assignment would have lead to deadlines being missed, which the author shows with a number of examples.

There is even more complex system then the ones covered by Audsleys algorithm. These are system where tasks have probabilistic execution times and an absolute guarantee of no missed deadlines can be given. Dorin Maxim et al [10] describe three sub-problems of finding the optimal priority algorithm in these types of scenarios where the basis is to find a failure rate as low as possible, i.e. the rate of expected deadline misses in the system.

2.2.1 Scheduling protocols

To make it easier to decide on execution order for systems with multiple threads, different scheduling algorithms has been developed over the years. Scheduling algorithms can work in different way but they all have in common that they try to do the scheduling as good as possible with respect to the information present at the settings of the system.

Scheduling algorithms can work as an offline scheduler or an online scheduler. Offline schedulers do the scheduling before system startup and stays with this scheduling during execution. Online schedulers use information during system execution to decide on the execution order.

Schedulers can either base its decisions on static priorities or they can use dynamic priorities. When a scheduler uses static priorities, all tasks priorities are set before system start. These priorities are used for scheduling decisions and all instances of the same tasks have the same priority. Dynamic priority based schedulers may have changed priorities for a task during runtime. Different activations of a task can have different priorities depending on the situation of the system.

Another difference between schedulers is if they use preemptive scheduling. When using preemptive scheduling, if a higher prioritized task gets ready to execute it gets switched in immediately at the next scheduling time. If a system is non-preemptive, all tasks executing gets to finish its execution before any new scheduling decisions are made.

There are a lot of different strategies for scheduling real-time system threads. One commonly used in RTOS is FPS [11]. FPS is mostly applied to tasks, and each task has a priority assign to it, which is decided before runtime of the system. The method of assigning priority on task-level is also known as "generalized rate monotonic". The task that gets to execute at a given time is the highest priority task that is ready to execute at that moment. This concludes to that all jobs within the same task gets the same priority [12]. A preemptive FPS is one of the most common ways of scheduling tasks in a RTOS.

If a system has hard deadlines associated with each task, a scheduling protocol like EDF could be used. Instead of using the priority, EDF lets the task with the closest deadline execute first.

Many of the existing more complex scheduling methods are based on either rate monotonic or earliest deadline first. Further developments of these were required to handle e.g. resource sharing [13]. John A Stankovic et al [13] mentions the need for handling e.g. periodic, aperiodic, preemptive and non-preemptive tasks in the same system. An aircraft is also mentioned as an example which system has 75 periodic and 172 aperiodic tasks, all with different requirements, in its control system [13].

2.2.2 Hybrid scheduling

The scheduling decisions aren't easy as different schedulers have different positive and negative aspects. Jukka Mäki-Turja et al [6] describes a way of combining static and dynamic schedulers so that a system can get the benefits from both of the schedulers. The technique presented uses a dynamic scheduler for event-triggered tasks and a static scheduler for time-triggered event, where hard deadlines are preserved for both the dynamic and the static part of the scheduling.

The authors take up an example where static scheduling is complicated to make. The example consists of the following tasks:

Task	Period Time	Computation time	Deadline
T1	10	2	10
T2	10	2	5
T3	50	1	2
T4	50	6	50
T5	100	8	100
T6	2000	7	100
T7	2000	8	100
T8	2000	8	2000

Table 1: Task description for Hybrid scheduling example

If this system is to be scheduled purely static, the developer has two choices. Either make a scheduler with a period time of 2000ms, which would make it large and memory consuming, or do a shorter scheduling pattern which results in a pessimistic system (T6, T7 and T8 would have to be scheduled more than once every 2000ms.

A better idea, given by the authors, is to schedule tasks T6, T7 and T8 with a dynamic scheduler while the other tasks use a static scheduler. The results from this implementation show that the tasks both use less total resources from the system and have better responsiveness.

2.2.3 Response time and jitter

Response time is the time that it takes for the system to produce an output for a given input. Response times can often be critical in hard real-time system and therefore of great interest to measure.

Response times could be both a single task execution and a series of threads executing and working together to perform a given task in the system. This type of response time is called end-to-end response time.

It is not only the response time that is interesting when talking about timings in real-time systems. As responsiveness and determinism is important factors for a system, the jitter is also a key aspect. Jitter is a deviation in time between different instances of a task or an occurrence in the system.

Jitter could be of different types in the system. Response time jitter is the deviation between the BCET and the WCET of a task. Another jitter is the deviation in activation time between instances of a task.

N. Audsley et al [14] presents formulas and calculations for determining bounds for both response time and jitter. Both determined bounds are of great use when scheduling tasks in a system. The authors then uses both bounds, amongst other properties, in calculations to schedule tasks for their presented scheduling technique, based on Rate Monotonic approach.

2.3 Common design issues

In real-time operating systems many problems can occur if not designed correct. Here are a number of common design issues that can ruin a whole system or at least make it not operate in a good way.

2.3.1 Task priority errors
When deciding on priorities for tasks it is important that the internal ordering of the priorities in the system corresponds to the actual priority between the tasks in the system. If priorities are set in an incorrect way, important tasks may get to little execution time. This may lead to errors in the system.

2.3.2 Race condition and memory errors
System consisting of multiple threads often has shared memory resources like static variables, lists and so on. These shared memories could be a reason for strange behavior in the system. The problem that may occur is so called race conditions [15].

Race conditions are when two threads, at the same time, are accessing the same memory position and try to manipulate it. In these scenarios execution orders decides the final results of the memory.

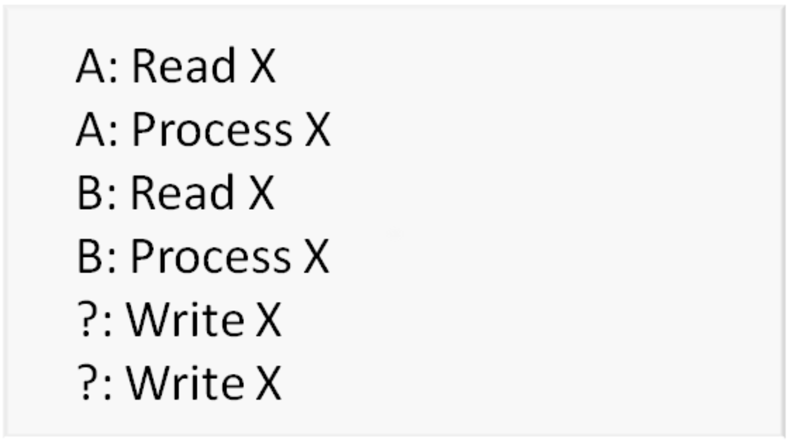

```
A: Read X
A: Process X
B: Read X
B: Process X
?: Write X
?: Write X
```

Figure 3: Race condition

Figure 3 shows a classic race condition. Both A and B are working on variable X at the same time. Depending on which order they update the variable, either the work from A or B will be discarded.

The solution to race conditions and memory errors is to protect all shared variables with e.g. mutexes. If a thread wants to use a shared memory, the mutex must be taken prior to the update. If someone else is working in the same memory, the thread has to wait for the other work to be finished before it is allowed to work on the memory.

2.3.3 Deadlock
A serious error that may occur in multi-threaded systems with a bad design is deadlocks. Deadlocks is a condition where two tasks have locked a resource and then waits for another resource before continuing the execution. If the two threads have locked the resource that the other thread is waiting for, none of the threads will finish the execution and release the resource. This means that both threads will wait an unlimited time for the resource and a deadlock has occurred [16].

2.3.4 Priority inversion

Priority inversion is a classic design problem in computer systems.

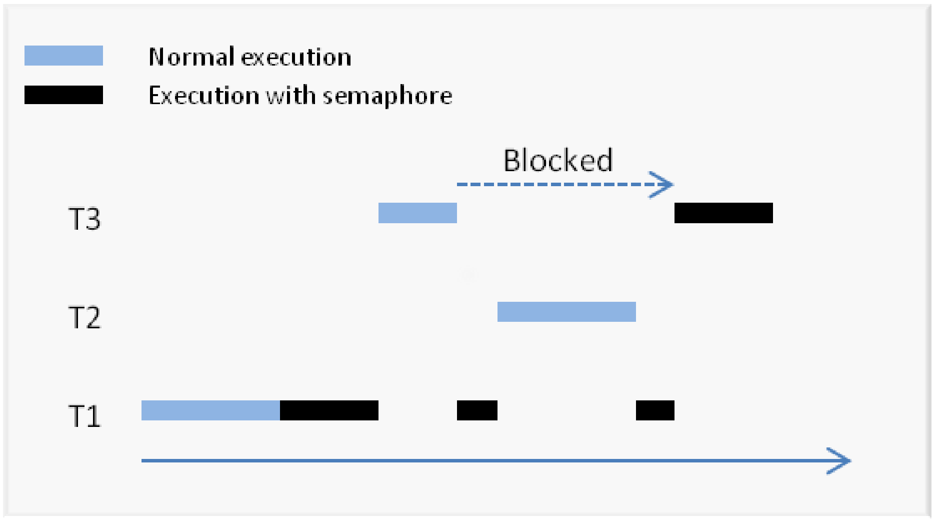


Figure 4: Picture of Priority inversion problem

Figure 4 shows a typical problem that priority inversion can give which can be really dangerous in a hard real-time system. Consider three tasks, T1, T2 and T3, where T1 is lowest priority and T3 highest. When T1 is executing it takes a semaphore in the system. It gets preempted by T3 that starts its execution. After a while, T3 also wants to get the semaphore and is therefore blocked by T1. T1 continues to execute but then T2 is ready to execute. Because that T2 has higher priority than T1, it is allowed to start its execution. Now T2 is indirectly blocking T3 to execute even though they have no shared resources [16].

A system behaving like this is highly un-deterministic and can cause serious execution problems.

The solution to this problem is to use a protocol to handle priorities of tasks in the system [16]. A widely use protocol is Priority Ceiling Protocol, PCP. It gives the task having a semaphore the same priority as the task with the highest priority that wants the semaphore. The protocol also prevents a task of taking a semaphore if another semaphore with a ceiling higher than the task priority is already taken.

Even though a protocol is implemented for solving priority issues the system can still suffer from bad design that makes the priorities in the system behave in a way that high priority tasks get to little execution time.

2.4 WCET analysis

The execution time for a task is the time it takes for the task to execute from start to end. The start is the time when it gets to execute and the end is when it has done its job and does not want the CPU anymore. This time will most likely vary with the input for the task. The worst case execution time, WCET, is when the task gets the inputs that take the longest time for the task to execute. There is also a best case execution time, BCET, which is a measurement of the time from start to end for a task with the input values generating the smallest execution time. The BCET is often not as interesting as the WCET when designing real-time systems.

Unfortunately neither the best case nor the worst case input are known in advance and are often hard to derive [17]. There have been a lot of techniques and tools developed during the years for estimating the WCET of a program, and many universities still do a lot of research on this area.

2.4.1 Problems with WCET analysis

When deriving a measure for the WCET of a system, a number of problems exists, this all must be solved to get an accurate result of the calculation. Reinhard Wilhelm et al [17] describes a number of problems and requirements that must be fulfilled for an accurate WCET analysis. First of all, all possible paths for an execution must be taken into consideration. Different input data and different system states can cause an execution to take different paths in the system which results in new execution times. It is important to catch all these different execution paths to not miss a execution that might lead to the WCET.

To show all possible execution paths, a Control Flow Graph (CFG) can be constructed. The CFG shows all possible paths in the system with the instructions associated with the path.

The next step is to exclude paths that never will be taken. This is done by doing a Control-Flow Analysis (CFA). The CFA examines all paths in the system to find execution patterns that will never be taken due to contradictions of the conditions in the statements. By removing infeasible paths, the result is more accurate.

2.4.2 Strategies for evaluating WCET

There are some common developed methods for deriving the WCET. There are two major interesting classes of methods for this purpose.

- **Static Methods**
  Some analysis tools don't use execution traces and analysis during an actual execution to evaluate timing on the system but instead the actual source code of the program to do its calculations. With the help of the code and annotations the static analysis programs can build up flow-graphs that show the possible execution paths with the defined values of parameters in the system. Combining these results with an abstract model of the target hardware the tools can achieve upper bound calculations for the program [17].
- **Measurement-based methods**
  Measurement-based methods do analysis by executing the actual code on the hardware, either the actual hardware or a simulation of it. With the help of the analysis the methods can derive timings for the program [17].
- **Hybrid Methods**
  A third method for analyzing a system is to use a hybrid analysis method [18]. *The hybrid analysis uses measurement for timing information of smaller parts in the system while a static analysis tool calculates the final WCET estimations from the source code. As these methods uses measurements for parts of the analysis they can both over- and under-estimate the final WCET*

depending on how the measurement has been made, and are therefore a bit less accurate then a pure static method, and are therefore not preferred to use in a real-time system with hard deadlines.

2.4.3 Methods for solving different tasks of timing analysis

Wilhelm et al [17] *presents a number of currently existing methods to solve the different problems. A timing analysis method uses a combination of these to calculate WCET.*

- **Static program analysis**
  Static program analysis builds on the static method with analysis doing on the program code.
- **Measurement**
  Deriving an approximate WCET by doing measurements is a good alternative for giving an approximation of the WCET in a system and is best used in non-hard real-time systems. The measurement might not be perfect but gives the developer a good picture on how long the task execution time is.
- **Simulation**
  Simulation based analysis is a good way to measure and analyze a program without using the actual hardware. By simulating the hardware and program simulation tools can get good results.
- **Abstract Processor Models**
  An Abstract Processor Model can be used when doing a static analysis to take the target hardware into account when making the analysis.
  Building a correct abstract model of a processor is not an easy matter. To have correct behavior of the model, correct information about the processor must be used in the model. The information needed is not always easy to get as manufacturers might not want to give complete information about important timings and features of the processor
- **Integer Linear Programming (ILP)**
  ILP is a language that is used to describe the system properties with the help of linear constraints. This method works best in just small code parts and not for large complex systems.
- **Annotations**
  Annotations are given to the analysis tool to describe different criteria's and settings of a system. With the help of annotation it is easier to derive bounds and features of the system in a way that makes static analysis possible. Examples of annotations are:
  - Variable bounds
  - Memory layout
  - Information about iteration and loop behaviors that is not explicitly explained by the code.

2.4.4 WCET calculation

It is possible to derive estimations of the WCET when combining methods listed above. The different methods provide their own set of properties for the derived WCET, and take more or less amount of time to execute.

Static timing analysis gives a WCET that is not an underestimation of the actual value. It can be called a bound calculation and is often an overestimation WCET. The bound can be determined by running an abstraction of the task on an abstract model of the target hardware. The abstractions do not contain all information and does not emulate the complete system correctly, e.g. cache optimization and other functionalities that might speed up the execution.

A common used method is dynamic timing analysis which tests a subset of all input data. This will derive minimal- and maximal observed execution time. Since the test only runs a subset of the data it will most likely not run the task with the exact data that gives correct BCET and WCET, and will most likely give a higher BCET and a lower WCET than the correct ones [17]. A development of this method is to calculate the same information on small parts of the task and then in the end combine the results to a result for the whole task. Even if this gives a better result it does not guarantee to find the exact times and it can lead to an overestimation of the WCET, if combining all the most pessimistic parts.

To take an overestimation of the WCET into consideration, when designing the system, is much safer than taking the estimation from the subset of input that might differ a lot versus the actual value. Although the dynamic result can give a feeling of how long time it takes and can be useful when creating a soft real-time system. It is also important to think about what data the task gets as input when doing the tests, e.g. if the input values that gives the WCET actually is an input that might happen in its natural environment.

2.5 System Debugging

A great help when debugging a real-time system when an error has happened is to have knowledge on the execution pattern and system states before the error state. To make this possible some sort of recording software can be used in the system. Hansson and Thane [19] proposed a method for system recording that can be used for multi-threaded and even distributed real-time systems. The method was to record system states and associate time stamps from a global clock with the stamps. With the help of the recorded information the execution could be reproduced again to see what happened prior to the error.

To get a better picture of system performance and execution some form of analyzing tool can be used. There are three main things to track and record for task execution [20]:

- **Identifying the task:** The first step of the analysis is to give identification for the task that is executing with the help of a task ID.
- **Time-stamping:** To make analysis of execution possibly, a time stamp needs to be taken on the places of the program that timing information is of interest
- **Reason for task switching:** Why did the task stop executing? Was it because of preemption by a higher priority task, waiting for a semaphore or simply that execution was finished?

2.5.1 Relevant system properties to monitor

An important question to answer is what properties to record in a system. The factors to take into consideration for this decision are the system resources used versus the ease of debugging when an actual error has occurred. More recorded information often gives the developer a better chance of reproducing the states and finding a possible error to a certain execution, but also gives more overhead during system execution. A small amount of recorded information on the other hand gives a smaller impact on the system but might not be sufficient enough to give the correct results during an off-line analysis and debugging. The first thing to think about during the implementation phase is what properties that exists that can be interesting to record [11].

- ***Response times***
  A key thing to record is response times in the system. This could be response times for a single task or end-to-end response time for a series of tasks that work together to do a specific job in the system.

- ***Jitter***
  An important property of a real-time system is jitter. There can be many types of different jitters in a system. A common variant is the difference in inter-arrival time for a task. Other jitters could be the difference between the BCET and the WCET of the task for example. If a system has high jitter the behavior of the system is less deterministic.

- ***Usage of system resources***
  The usage of different system resources is interesting to have as a basis for evaluation of a system. This resources could for example be the CPU usage and usages of a shared communication line or similar.
  Variables and logic resources can also be logged. If a variable is accessed and changed globally, it could be easier to add some kind of sampling of the variable at specific times; instead of saving the value of the probe in each and every place the probe gets a new value.
  Queues and buffer can be monitored by adding a callback or a new function call in the wrappers that get and put data on the queues or buffer. It could also be interesting to measure how many elements that exist in the buffer or queue and could also be done by adding a simple integer probe.

2.5.1.1 Task switching

Task switches often occur frequently and is often a major source of information of what went wrong. Which task got preempted, why did it get preempted, which got to run instead and how long has the task been running are questions that you can get an answer to if incrementing the task switch functionality of the operating system. In VxWorks this is done by hooking up a simple callback function that gets called with necessary parameters every time a task switch occurs.

2.5.2 Probing and the probe effect

To measure the time between different jobs in the system, measure points needs to be inserted in the code. This way of measuring the system is called probing. One probe is placed in the beginning of a job and one probe is placed in the end. By measuring the difference in time between the executions of the two probe lines a job time is achieved [11].

If probes are added to a system for measuring its behavior, the system will be affected by these probes. First of all, the overall execution time of tasks will increase as more code has to be executed. Task switches will take longer time because of the overhead from the recording software. This will also increase the interrupt latency on the system as no interrupts can be processed during the context switching. What this means is that the system will behave differently when probing then it did before the probes were added [2][11].

If a system is monitored with probes during development and implementation and then gets its probes removed in the final version of the system, the measurements done during implementation will be wrong as they measured a different system. It could be the case that the extra code presented from the probes made the system work in a different, more correct, way. Because of this, a system evaluated with probes should have the probes still running in the code of the final version. In this way the system released will be identical to the monitored system and the properties measured in the system are valid for the final system also [11].

2.6 Analysis tools

There have been heavy developments of tools for analyzing and visualize scenarios of real-time systems during the past few years. Almost all big companies that provide a real-time operating system also provide some sort of analyze utility specific for their product.

2.6.1 Trace recorders

To collect and save real-time data from a system some sort of trace recorder is used. Trace recorders often works with a circular buffer that continuously stores information from present time and backwards a specified time. The information stored can later be used to evaluate and investigate a system to find parts that doesn't work as planned.

There are a number of key factors when deciding on how a trace recorder should work:

1. What, and how much, information is necessary for the analysis? More information gives better analysis possibilities but could interfere more with system execution.
2. How long time is interested to store in the buffer? More execution time saved allows the user to trace executions further back in time but uses more memory of the system.
3. How easy is the recorder to modify and use? A good feature of a recorder is to easily be able to customize the recorder to fit the needs of the system and developer.

A well working trace recorder should be able to run in background of normal execution with just small CPU load in the system. The load must be so small that it does not change the behaviour of the system.

2.6.2 Offline analyzers

The information from trace recorders is often just raw data that is hard to understand for a tester. Since the nature of these trace recorders is to use as little memory as possible, the traces will be compact and hard to read manually. Therefore some kind of interpreter is useful which can present the information from trace recorder files in an easy and understandable way. It is important to use this data to present relevant information in a way that is easy to draw conclusions from.

The interpreting software can also include smart algorithms to identify states and give information that is not obvious by just studying a log text file. This helps a lot when trying to identify problems and erroneous states in the system.

3 Evaluation of Existing tools

To decide on further work in during the thesis, a number of analysis tools were examined to find out if there are any currently existing tool that fullfills ABB's requirements. After searching for tools, three third party tools and the ABB tool JobMon where chosen for further investigated. The three third party tools are Tracealyzer, System Viewer and TraceX. In this chapter a short summary of all tools and their features is explained. This information is later used in the selection process in the thesis.

3.1 Tracealyzer

Tracealyzer is the name of software package that can record and analyze sequences of events in real-time operating systems developed by Percepio [21]. It consists of two parts; the embedded recorder and the graphic offline analyze tool.

3.1.1 History

Tracealyzer was from the start a research project at MDH developed by Johan Kraft. He worked together with an industrial company to develop a recorder and a graphical interpreter during his PhD thesis [20]. To help understanding Tracealyzer and its advantages better, a meeting with developers at this company was made during the thesis.

3.1.2 Tracealyzer and the company

The company is using Tracealyzer and its trace-recorder in their products and the recorder is even enabled during normal operation at their customers. In the meeting representatives from the company explained how they have implemented and used the recorder online in the system and what help the analyzer has been in their work.

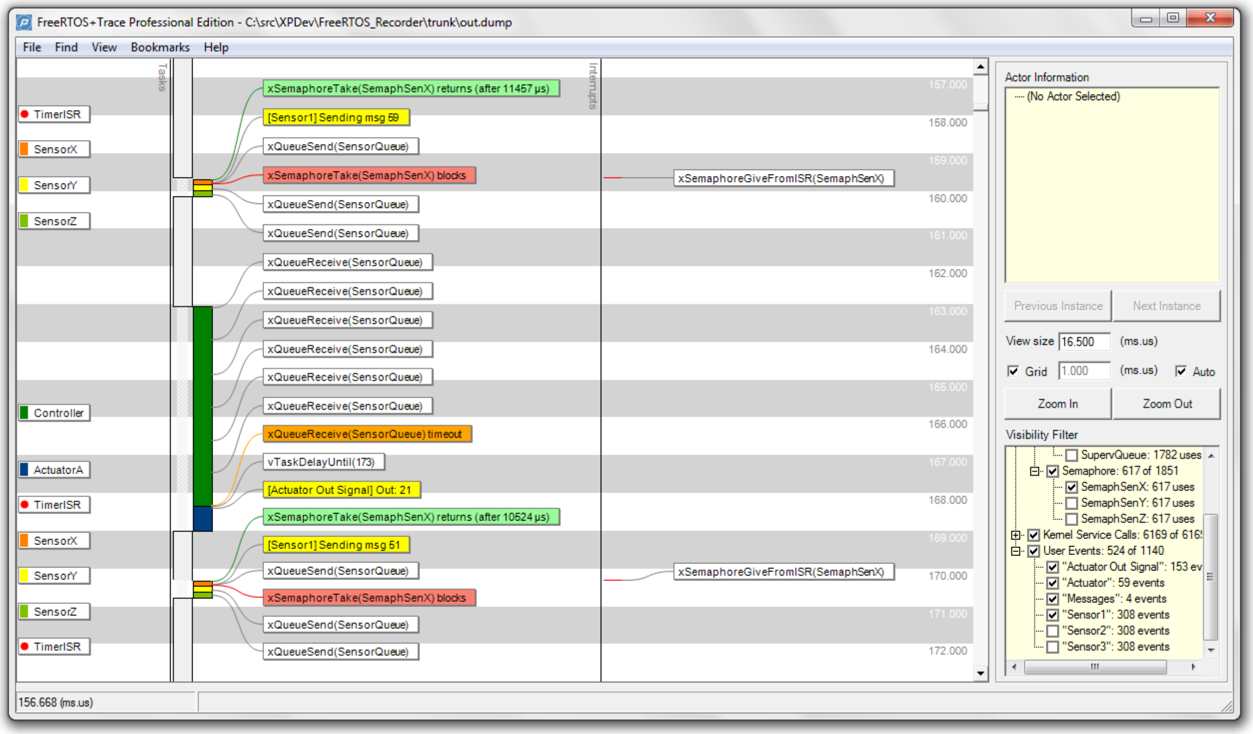
In their complex system, a number of system recorders are used, where one is the Tracealyzer trace-recorder. All this collected information is supervised with a maintenance-class that takes care of the snapshot taking in the system. Snapshots of the system are taken at specific system events defined from the company, where the information is stored locally on the product computer.

When the company personal wants to investigate a log they can download the recording files and open them in the Tracealyzer tool. As the company and Tracealyzer developer Johan Kraft cooperated during development of Tracealyzer they have got the analyzer custom made so that it can open and merge the information from both the recorders and the product-specific monitors and recorders.

3.1.3 Tracealyzer today

The software has changed a lot since the company implemented the first version. It has been commercialized and is now a property of the company Percepio.

The first part of the software is the recorder. The recorder is a small program that is open-source for the paying customer. It is integrated in the product and continuously records data of the execution with the help of ring buffers. The events recorded can be e.g. task switches and semaphore give/take and each event includes extended information. For example a task switch event is extended with why the task switch event happened, who was running and who runs after and when this happened. All this is done during normal system runtime. The time stamped events are put in RAM for later upon system failure or other trigger be saved to a file. The recorded data takes around four byte per event.



xSemaphoreTake(SemaphSenX) returns (after 11457 µs)
[Sensor1] Sending msg 59
xQueueSend(SensorQueue)
xSemaphoreTake(SemaphSenX) blocks
xSemaphoreGiveFromISR(SemaphSenX)
xQueueSend(SensorQueue)
xQueueSend(SensorQueue)
xQueueReceive(SensorQueue)
xQueueReceive(SensorQueue)
xQueueReceive(SensorQueue)
xQueueReceive(SensorQueue)
xQueueReceive(SensorQueue)
xQueueReceive(SensorQueue)
xQueueReceive(SensorQueue) timeout
vTaskDelayUntil(173)
[Actuator Out Signal] Out: 21
xSemaphoreTake(SemaphSenX) returns (after 10524 µs)
[Sensor1] Sending msg 51
xQueueSend(SensorQueue)
xSemaphoreGiveFromISR(SemaphSenX)
xSemaphoreTake(SemaphSenX) blocks
xQueueSend(SensorQueue)
xQueueSend(SensorQueue)

(No Actor Selected)

View size 16.500 (ms.us)
☑ Grid 1.000 (ms.us) ☑ Auto

☐ SupervQueue: 1782 uses
☑ Semaphore: 617 of 1851
☑ SemaphSenX: 617 uses
☐ SemaphSenY: 617 uses
☐ SemaphSenZ: 617 uses
☑ Kernel Service Calls: 6169 of 616
☑ User Events: 524 of 1140
☑ "Actuator Out Signal": 153 ev
☑ "Actuator": 59 events
☑ "Messages": 4 events
☑ "Sensor1": 308 events
☐ "Sensor2": 308 events
☐ "Sensor3": 308 events

156.668 (ms.us)

Figure 5: Tracealyzer graphical tool [21]

Tracealyzer includes an advanced graphic offline tool for analyzing the files that get written by the recorder. An example view from the tool can be seen in Figure 5. The tool can read a file that is dumped by the recorder and replay all events in a graphical time lined order. The authors have made a vertical time line in difference from the horizontal view used in e.g. System Viewer. The timeline for tasks makes the user able to go back in time and see what actually happened and why it happened. The main view of Tracealyzer shows a time line with all the active tasks and how they run and preempt each other with additional information to be expanded. There are also lots of different sub views; CPU load, semaphore history, kernel calls, user calls and more.

The different views and windows of the Tracealyzer are linked together so that selecting one event in one window shows the same event in another window. This could be used to see different aspects at the same time on a specified event of the system. One useful case would e.g. be when showing CPU load at a certain time point. The user can click and it will zoom into the specific point where this happened on the task time line. This makes the user able to see what actually happened, task-wise, when e.g. a CPU load spike occurs.

3.2 TraceX

TraceX is another commercial tool for system analysis [22]. The tool is developed by Express Logic in its main focus is on the operating system thread, also developed by Express Logic.

Features of TraceX:

- Automatic priority inversion detection and display.
- Built-in execution profile report that shows system usage of the different threads.
- Stack usage on a thread level for the threads loaded in the analysis software.
- Raw trace dump that can be read in for example Notepad.
- Multi-core support.

TraceX is built for use on ThreadX's own real-time operating system, and there is no information if or how good it works with VxWorks.

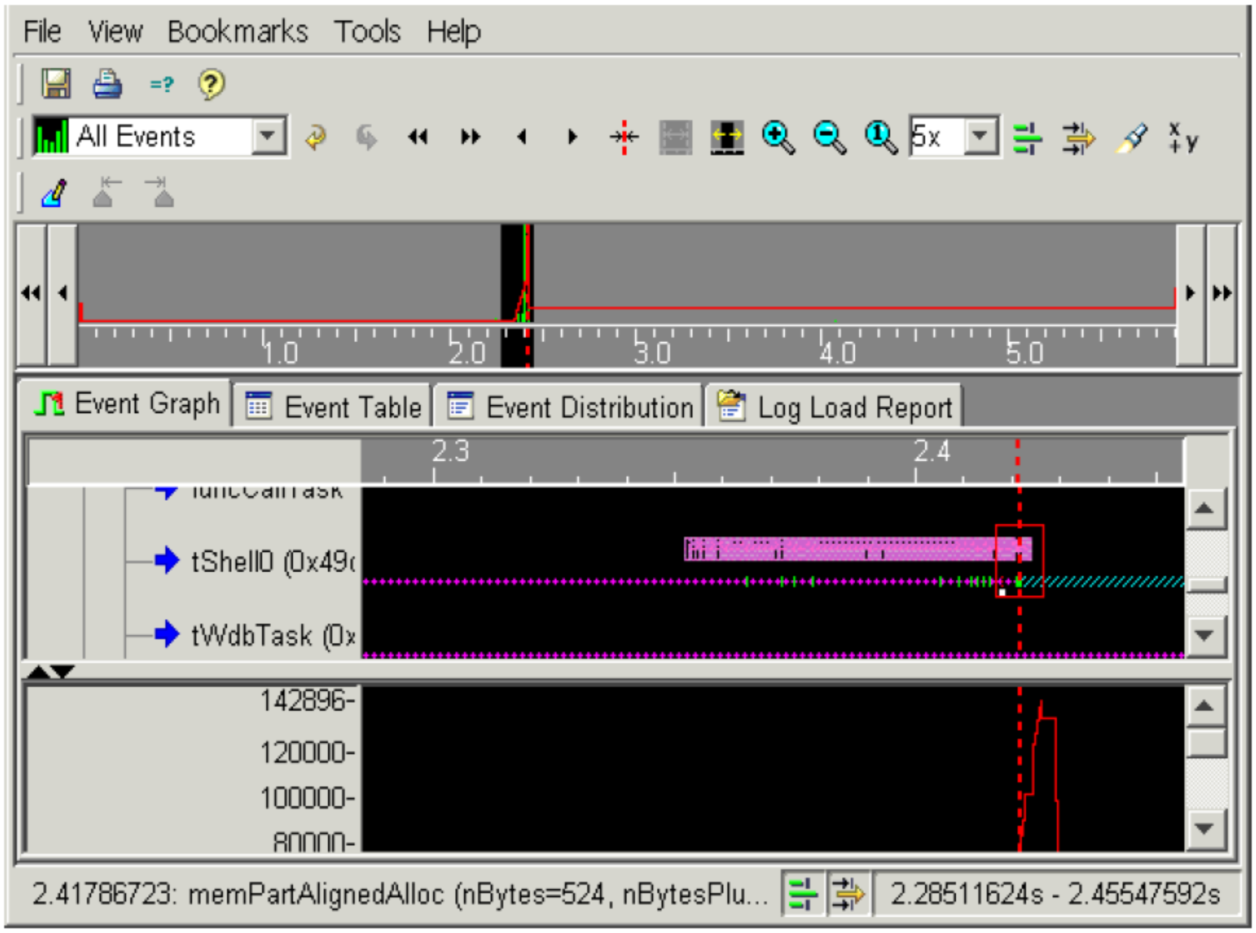
3.3 System Viewer

System Viewer is a further development of Wind River's System Viewer [23]. It comes with all tools needed to trace an embedded system both on the run and offline after a log file has been created. In the recording mode – for offline analyze - the tool has a lot of functionality in common with Tracealyzer.

Wind River's System Viewer can be configured to continuously write events and information into ring buffers. It can be triggered by an event to write the buffer either to file or upload the data through one of several protocols supported. The collected information is basically the same as Tracealyzer and System Viewer also comes with an offline tool to analyze the created log files.

The user can determine which events and system calls will generate a trace in the log file. System Viewer's recorder hooks into the system and will write all necessary information for context switches, semaphore actions, interrupts and more if wanted. The information is often just a timestamp together with the involved task(s) and takes a small amount of space. Of course the more information the user chooses to save in logs; the more CPU Load on the system and the more memory used by the recorder.

The recorded files can then be opened in a graphical tool, shown with an example picture in Figure 6. The tool presents all information based on a horizontal timeline. It is then easier to get an overview of the system than reading plain text in a log file. The graphical tool will display all events logged together with the extra information saved on each event.



2.41786723: memPartAlignedAlloc (nBytes=524, nBytesPlu...

2.28511624s - 2.45547592s

Figure 6: System Viewer graphical tool [23]

The extra load of the system is not well documented in System Viewer's manual and therefore unknown.

Since System Viewer is created specifically for VxWorks it is also able to perform things like creating log files after a warm reboot. The VxWorks kernel can be configured to not erase a specific part of the memory on a warm reboot. This makes System Viewer recorder able to save the logs in a memory that does not get erased and therefore it will be able to write a log file with the system history leading up to a crash on next boot [23].

3.4 JobMon

JobMon is an analysis tool currently in development at the company. The idea of JobMon is to monitor and give information about current jobs running in the system. It was developed to work as a help when analyzing the system and to get timing information for important jobs in the system.

3.4.1 System events

Today, JobMon focuses on five events that happen in a job.

- **Trig event** - A trig event is the first event that happens that requires a start of a job. This could be an external signal, a time-event for a periodic task etc. and gives the job a signal that it should start
- **Schedule event** - The schedule event is when detection is done that there is a need to start the job processor
- **Wake event –** This event marks the start of the job-specific code
- **Response event –** The first response from the job, e.g. the first response byte sent
- **Done event –** the job-specific code has finished executing

3.4.2 Job monitoring

A job is not a specific task but more a series of different events in the system which reacts and response to an event. This event could e.g. be an analog input to the system and the response could be a triggered break of the line because of an error. The reason to monitor the system on a job-level and not a task-level is that the important times in the system is the responses to system events and not how long an actual task has executed.

The primary function of JobMon is to monitor the system on a job level, a form of end-to-end response time. A job is a series of actions done in the system to give a response to a specific input. The input could e.g. be an analog input to the system and the response could be a triggered brake of the line because of an error. The times for the system to respond to inputs are critical and therefore also the time a job takes.

The main information stored in the JobMon object is a number of time spans. These times are measured by adding JobMon calls in the system where the specific part of the code has been executed. By measuring the time between these events, different times within a job is calculated. The system saves seven different time intervals. These are schedule to schedule, schedule to wake, trigger to response, trigger to schedule, trigger to trigger, wake to done and finally wake to wake. For each of these, the two time stamps, calculated time for last execution, minimum execution time, maximum execution time and time variance is saved. No logging is done for older executions except these timings.

To see the information a dump-command is written in a terminal which triggers a print of all times for the different jobs. This requires that an observer is continuously running this command at interesting points in the system to get the relevant information from the tool.

3.4.3 Thread monitoring

To monitor the system on a thread level, JobMon consists of a thread monitoring part. The thread monitor hooks on to tasks and when a context switch happens, a defined method is run. By logging which tasks that gets to run and which who got preempted the monitor can give relevant information regarding behavior on a system level.

It is possible to connect one thread monitor object to a specific JobMon object. This could be used to get further information about the job, like for example what was the last task that preempted the job. This is only useful for the case where one job is just one thread. For cases where jobs have multiple threads it might not be as interesting to log just one thread execution.

The implementation today doesn’t use any recording so the information that can be given from the monitor is number of context switches, last preemptor as well as timing for last execution and information about maximum and minimum execution in ticks and time.

3.5 Selection process

During the thesis work, a theoretical survey of all three applications has been done. The authors of this thesis have met the developers at a company, using Tracealyzer, during the project and they have given their view of it and how it helped them. Johan Kraft and a colleague from Percepio have also visited us here at ABB for a presentation of what Tracealyzer can do and showed a short demonstration.

The product looked for is something that can write a log file upon a system error or whenever specified by the developer. The log should contain enough information to have a chance to solve the problem and a graphical interpreter of the log file is therefore a must. All three, System Viewer, Tracealyzer and TraceX, have a smart graphical user interface but Tracealyzer is pushing that they have an even smarter interface and easier to use. A small survey among developers at ABB shows that many find System Viewer hard to work with and that it has a complicated graphical interface.

3.5.1 Available options

After doing a theoretical investigation on current analyzing software and ABB demands, three main alternatives for analyzing software has been worked out. The three alternatives are:

1. Developing and using JobMon only.
2. Using a new version of JobMon in combination with Tracealyzer or System Viewer.
3. Using Tracealyzer or System Viewer without JobMon.

These three alternatives will be compared in the next section to draw a conclusion on which alternative that best suites the needs from ABB. There will also be a comparison between Tracealyzer and System Viewer to see which of these two tools to choose if the conclusion is to not use JobMon as standalone analysis software.

3.5.2 Options discussion

The framework ABB want in their products will probably never be found on the existing market. Both Tracealyzer and System Viewer are developed for the purpose of monitoring a system and debugging either a pre-defined sequence or a sequence where you suspect an error. There is no way to setup limits or other features that can trigger a log at specific condition.

System Viewer offers an online debug view where you can run the system normally and monitor all information on the run. This is a good feature, but when you do not know if, when or where an error might happen, this way of debugging becomes exhausting. Many developers at ABB who have worked with System Viewer think that it has an complicated graphical interface and is hard to use. The tool is not used every day and therefore it is a must that it is so easy that you remember all common functions between the occasions.

From what Tracealyzer and System Viewer specifies for the public they theoretically fulfill the same purpose seen from this thesis work's perspective. Both System Viewer and Tracealyzer offer system logging where all events are logged into a ring buffer and saved to file when something triggers the save function.

The logs made by both tools would probably be enough to find most errors in the system, but it is not possible to specify what an error is.

A large industrial company has, as already stated, implemented Tracealyzer in their product control systems. The major difference from our point of view is that there already was functionality to detect system failures. This means that the trigger to write the log file already was implemented before they even thought of Tracealyzer.

The framework for specifying a system error is specific to each system, therefore no such implementation is made in neither of the tools. Each system has their own set of errors, e.g. buffer overflow, deadline miss and/or erroneous sequences of executions. This concludes to that something system specific needs to trigger the write function of the loggers upon a detected system error.

JobMon, which already have some basic functionality, is developed in the purpose of detecting system errors. Today it also has some functionality for logging system and some thread events. The error detection is limited to a monitoring part with time between events. There is no alarm functionality implemented and the system logs collected by JobMon are limited with no way of writing them to a file or analyzing them in a graphical offline tool.

Review of the options above:

1. Developing and using JobMon only.

   Possible, but would take a lot of time. It would not be possible to, during this thesis time, develop a fully functional graphical interface to interpret the logs written by a recorder.

2. Using a new version of JobMon in combination with Tracealyzer or System Viewer.

   Possible and would not take too much time. JobMon will serve the functionality of an evaluating- and error detecting-framework. Tracealyzer or System Viewer would fill the logging and log interpreting functionality.

3. Using Tracealyzer or System Viewer without JobMon

   Not possible without custom designing Tracealyzer or System Viewer. It is impossible for the standard tools to recognize error conditions in a specific system. Logging and debugging functionality is useless if nothing gets triggered to write the logs from RAM to file.

3.6 Discussion

The solution to this specific problem could be cooperation with e.g. Percepio (developing company of Tracealyzer) to custom design the Tracealyzer recorder to be able to measure several properties that can indicate a system error. Exceptions in time between events, value of a counter, number of elements in a buffer, or other developer specified error would trigger Tracealyzer recorder to write a log file for debugging offline.

Another solution, and or suggestion, is to extend JobMon and make it the system-fault trigger component - the system that triggers the real system logger to write a log file. This would work with both Tracealyzer and System Viewer, whichever the company chooses, it is probably a question of cost vs. easiness. Since it is not possible to test Tracealyzer, there is just a possibility to review the specified functionality of it.

It would also be possible to develop an own trace recorder and a graphical interface to interpret the log files. But this would take too much time, especially for the graphical interpreter, to fit within this thesis timeframe.

JobMon is already a powerful tool and can with some effort be extended to be able to trigger the log writer. This would help the system developers by having a log file of the past seconds leading up to a defined state interpreted as a system error. The information in e.g. Tracealyzer is extensive and would probably be enough – together with a small JobMon log – to understand the error and debug the system. JobMon can also easily be extended to include any information missing in System Viewer's or Tracealyzer's log. This might be some system specific information.

The new version of JobMon must a fulfill a couple of requirement to be usable in the future

- Must not change the behavior of the system in any way
    - Must not increase the CPU load noticeable
    - Must use small amount of memory
    - Must never be able to crash the system – always "passive". Exceptions in JobMon must always be treated and must never interfere with the other system.
- Must be easy to setup criterions interpreted as system error (e.g. time between specified probes).
- Must be able to take an easily specified action on system error.
- Could save a little dump of its current information on a user defined error state, e.g. which alarm that trigged the dump.

4 Case-Study Implementation

4.1 System architecture

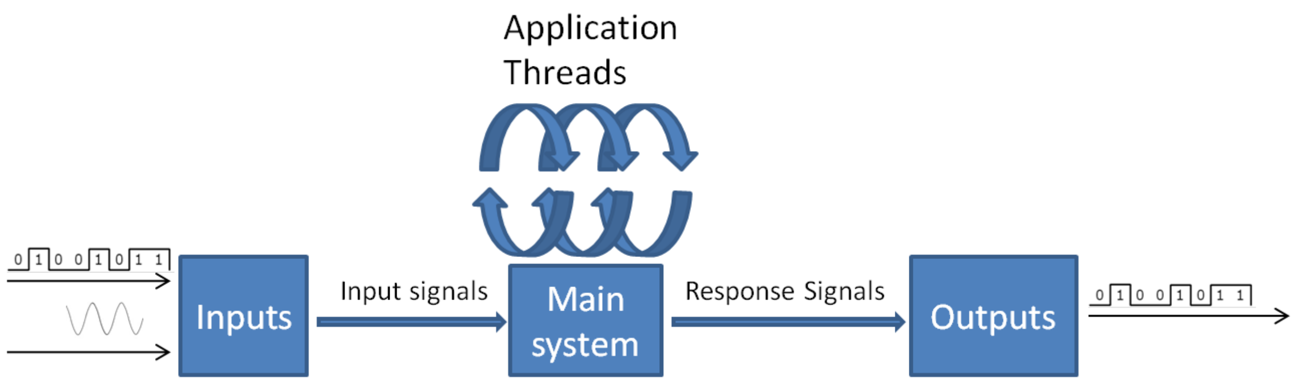


Figure 7: System Architecture

The hardware consists of a lot of binary and analog data inputs, a motherboard with CPU, RAM and flash and components for output signals. It is I/O driven and the input data is measured and calculated in a long chain of executions. After a lot of calculations on the data, an output is produced to an actuator in the end.

The CPU has a clock frequency in the range of 600MHz and produces around 70 million system ticks per second. It is important to know a bit of the CPU when interpreting tick results and other data from our measurements.

4.2 Software setup

The implemented system consists of over a million lines of C++ code. Therefore the implementation of new additions to the system is not so straight forward. It is a must to understand the core functions in the system and to reuse already existing optimized classes, e.g. double linked lists. It is also important to use the same pattern for writing code as previous authors to make the code easier to understand and perhaps extend or change in a later phase by someone else.

4.2.1 ABB Real-time system execution model

ABB has developed a complex model for executing many threads and components concurrent in their system. They run a normal VxWorks priority based scheduling for the threads, but the system can be divided into two types of system execution scenarios.

In the first scenario there is an internal way of scheduling small parts of the task, called components. Each thread that uses this type of execution pattern have components inside that all have inherited from a base class. This base class provides an interface to be executed in a structured way within the same thread. When the thread gets the CPU it starts to execute components in a pre specified pattern. Each component has an integer that specifies when it should be executed within the thread.

The other way of executing is a more common way with pure threads that have their code in the body.

Since each thread is assigned a priority at start, it is important that all code executing within the thread are equally important. It is not possible to mix priorities within the threads, since the scheduling of threads is done by VxWorks and will interrupt all lower priority threads upon request. The hard part is to break up the system and group the code that are equally important and be sure to not mix in something that has no time limit or other that can be executed at a lower priority.

4.2.2 Component inputs and outputs

Every component executing within a thread automatically inherits properties for using inputs and outputs through a complex system specific wrapper. The use of this is to be able to communicate between components in a simple way. An output can e.g. be bound to an internal variable and through the interface be read by any component.

There is also a possibility within this complex framework to import settings and parameters from the database and bind them to internal variables. This is frequently used to configure different settings within the code to avoid hardcoded limits, settings and parameters.

4.2.3 Lifecycle management

It is important, for every object in the system, to have lifecycle management. This is due to the target system is dynamic and e.g. services and protocols can be taken down during runtime and then get re-initiated at any time. This puts the same constraints on our implementation; all objects have to be able to get created and destroyed properly during runtime.

The hard part with lifecycle is if an object can get destroyed it must be sure that no one else is interested in this object or that it has some kind of reference that it might use later on. Use of a reference to a memory address, that is not allocated or allocated to a new object, will most likely end in system wide disaster.

One strategy of solving the issue of lifecycle management is to use reference counts. Reference count is a simple integer implemented in an object that increases or decreases when other objects announce interest in the particular object. This is a must when e.g. saving a pointer to the object. An object can therefore never be deleted until the reference count is zero. The developer can then always be assured that the object exists whenever an increase reference count method call has been done and the integer inside the object therefore is greater than zero.

It is also important to always be assured that no objects get to hang loose. This could be the case if someone does not decrease reference to a specific object it has increased the reference count of earlier. This would make an object live forever, since it will never reach a reference count of zero.

4.2.4 Locatable objects

The system today provides a locatable object interface. This means that the whole system has a common namespace where every object registered in lookup table is able to be located. This makes communication between different objects possible. It does not matter if they are not related in any other way as long as they know each other's name – or OID, Object Identifier, as they are called in the system.

Locatable objects have a common interface inherited from the base class. The class includes the read and writes functions which are the specification of the external interface towards other objects in the system.

4.2.5 Job description

ABB system is built up around series of executions of threads that combine their calculations and functionality to complete a job in the system. A job does not have to be a single thread but can stretch across several. This makes thread monitoring not so interesting and the ability to measure properties for a job extremely interesting. The error detecting system must therefore support probing and measuring this concept. It must be possible to specify start and end points that are separated both in code files and in threads.

5 SysMon Framework

A new tool will be developed with similar measuring properties as the old JobMon. The problem with JobMon is that it is written in a static way, allowing just time measurements. It is also limited in how many time measurements it can do, and uses static naming for all measurements. There are static functions for calculating time delta and no current functionality for setting alarms or similar warnings when time passes a pre-defined limit. There have been thoughts about making warnings or alarms for a long time, and by studying the class diagram one can see the functionality has been thought of, but not yet implemented.

The new tool, SysMon, must be able to measure any kind of data. It must be so generic that it should be possible to implement measurements later on of data types that are not even known when developing the framework. The only data that are known about right now is time and the focus during this thesis will be time measuring – without limiting anything to just handle time.

The thought is to create the basic framework and prove that it works by measuring time, and thereby also making the old JobMon redundant. It is also important that the tool and code is easy to understand and simple for the developers at the company to start using and develop even further to fit all their demands.

The old JobMon also implemented support for monitoring on thread level, e.g. CPU load for a thread. These functions can be included in the new SysMon framework as they are, but will not be a priority since there are thoughts to implement an external logging utility that can save logs of the system history. These logs should provide enough information of what went wrong, and most likely more useful than what JobMon presents on thread level today.

5.1 Development plan

To satisfy all demands from ABB, JobMon will be developed further and the class structure of it will be changed. A good implementation way of SysMon is to do it in an incremental way taking and a new step when there is time and the last phase is finished. An idea of development steps is:

1. Setting up a framework that puts as few constraints as possible of what can be measured or evaluated. It must be possible to measure and evaluating complex scenarios as well as a simple integer counter. The classes created will be one probe class that stores the information at specific system events, and one measurement class that can correlate and perform calculations on data from probes. From these base classes the developer will be able to derive inherited classes can perform any type of probing and any type of correlations between probes in the measurement object.

2. Implementing special classes that can be used throughout the system. Example of probe types are:
    - Time probe - saves timestamp from specific places or events in the code
    - Tick probe- saves real system tick counter as integer

    The probing functionality must be thoroughly tested to make sure that it doesn't interfere too much with the system and also that it doesn't produce any bugs or errors in the rest of the code. Time interference of probes can be tested by probing the probes itself and using time delta measurement to evaluate the difference in time.

3. A probe on its own does not provide any useful information. Therefore there will be measurement classes for the most common measurements in the system, e.g.:
    - Time delta - measuring time between two probe timestamps
    - Tick delta - measuring ticks between two probe stamps

4. Integrating a framework for setting alarms on the evaluated measurements. This framework must not set any unnecessary constraints on what can be seen as a faulty state of the system. It must be able to perform simple compares as well as advanced mathematical functions.
    There also must also be a way of populating the alarm framework and set the correct parameters for every object in the system. This must be done either through an XML file or by using the database directly.

5.2 The framework

The most important part of the framework is that it must be as general as possible and not obstruct our further development. It must also be easy for the people working with development and perhaps using our tool to extend and implement specific measurements of the software. Another important factor is to make the code slim in a way that it doesn't bring a lot more CPU load to the system.

The framework will be built in an object-oriented way with generic base classes that handles the basic functionality with possibility to extend the classes to special implementations in an easily. It is focused around three main requirements, where each will have its own code part in the system. The three parts is the probing of the system execution, measurements based on the values from the probing and evaluations on the measurements.

The main thought with the framework is that it should build a stable ground for all kinds of measurements. The developer should then be able to inherit from the base classes when creating a specific measurement and by that only have to implement the special functionality of the specific measurement. All other management and functionality is already implemented in the base classes.

The three base classes are:

- **Probes:** Probes stores information gathered from the system.
- **Measurements:** Measurements uses one or many probes to calculate a result from the information gathered
- **Evaluations:** Evaluations uses the results from one or many measurements to decide on possibly actions in the system or alarms that the measurements should give

An analogy to the real world can be done by this comparison:

- **Probes:** Giving the current temperature outside.
- **Measurements:** Calculate the average temperature, maximum temperature and minimum temperature during the day.
- **Evaluations:** Decides if it is hot enough to go swimming today or not, depending on the result from the temperature calculation.

5.3 Architecture

To be able to find and use already existing probes and measurements and to handle the memory usage, some sort of management-class is needed. It is also necessary to divide information between different parts in the system in a way that all monitoring information for one specific part of the system is easy to find and edit.

The start point was to create a class called “Manager” that is supposed to keep track of objects that are interested in each other. There will be one instance of the manager for each part of the system, for example a transfer link, a runtime thread and so on. To fulfill this, the manager will be a locatable object in the system and have a unique name identifier. The different parts of the code that belongs together can then connect to the same manager and therefore access the same measurements and probes.

The manager will keep track of all measurements associated with it, and all probes associated with its measurements. It is also from the manager object the developer requests the correct probe pointer for setting a value and doing measurements on the probes.

5.4 Use cases

There are a few main use cases that are important for the developer to know about. A use case picture is presented with the basic uses of SysMon and shows what possibilities the developer has to interact with the tool. A more complex and detailed description of the use cases will follow later in the report.

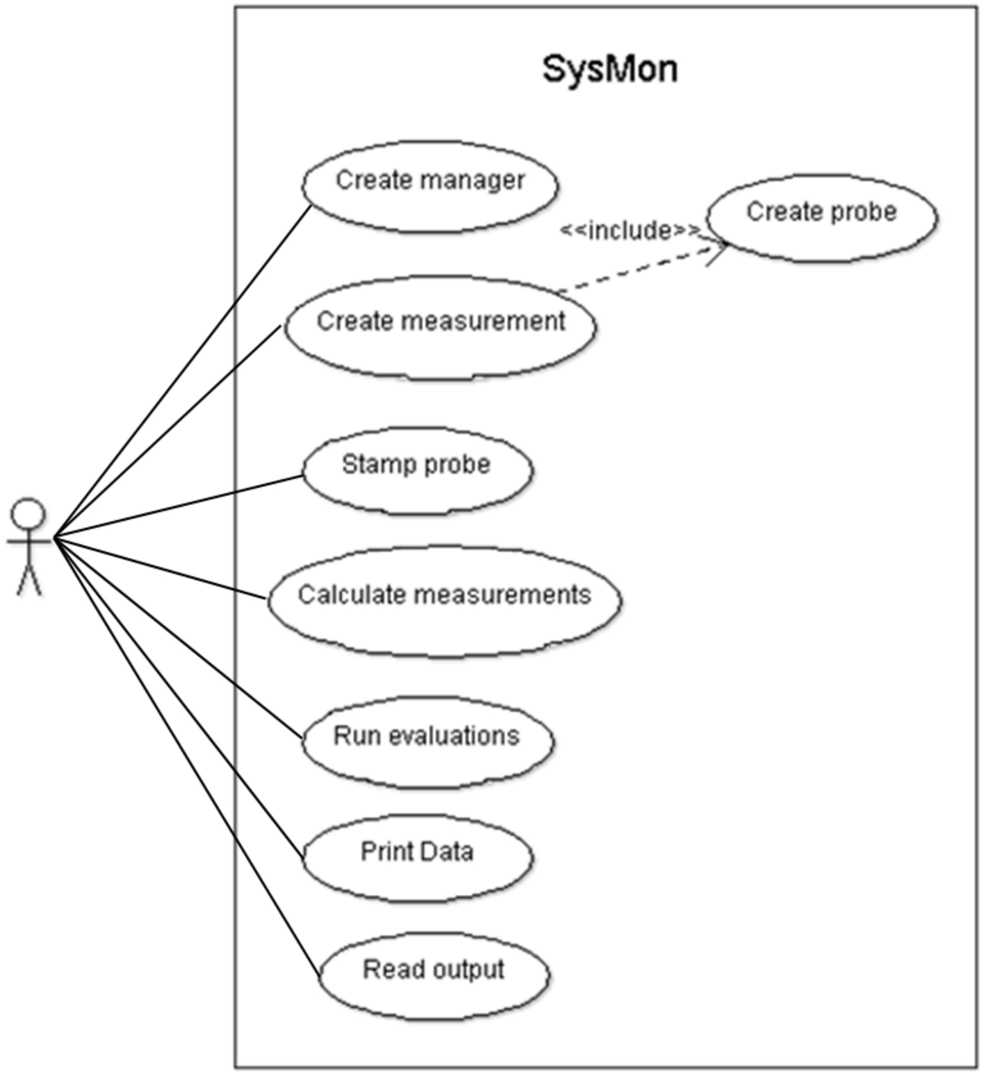


Figure 8: Use Cases

The use cases are focused of fulfilling three important requirements of the system. These are the possibility to take measurements, evaluate them and showing the results to a developer. To take measurements there are use cases for creating and initializing managers, probes and measurements, and to stamp probes. The evaluation part is the calculation of measurements and running of evaluations. The last two, print data and read output, is to get the information either to a text window or by having some sort of output signals from the tool.

5.5 Conceptual class diagram

From the class specified, a conceptual class diagram has been constructed.

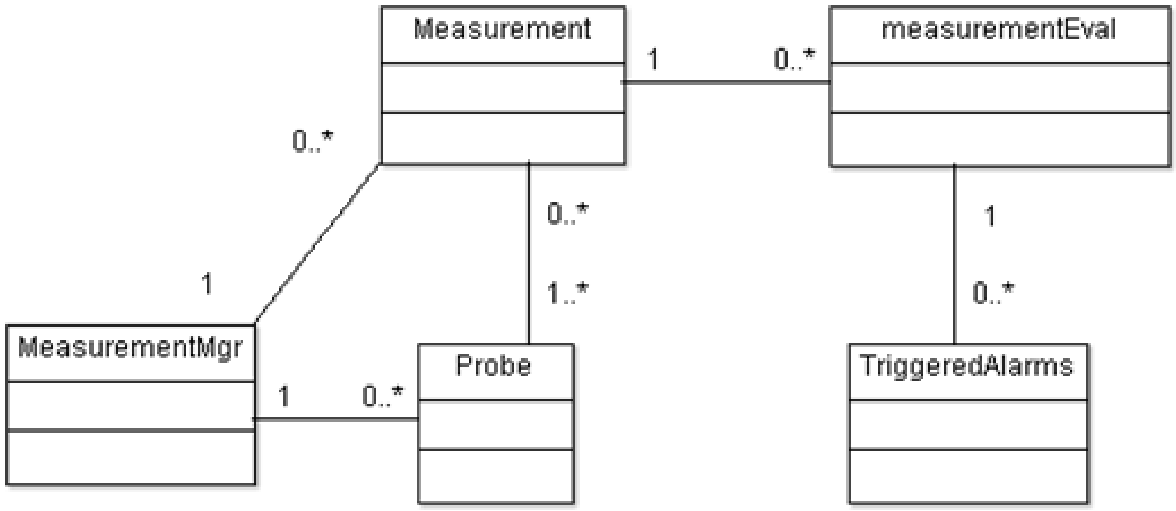


Figure 9: Conceptual Class diagram

The main functionality for the design is show in Figure 9. These classes cover the basic functionality aimed for in the final products and how they are associated to each other.

From the system description given earlier one can see that a new class has been constructed, the TriggeredAlarms class. This class is used for recording history data regarding results from the evaluator that has given an alarm state in the system.

5.6 Implementation details

The pattern of coding is taken from the earlier implementation of JobMon. The class methods, naming conventions and code style have been adapted to be easier for a regular developer to continue. The small details of implementation will be skipped and the continuing parts will explain specific parts in a high abstraction level.

5.6.1 Alarm handling

The evaluator class task is to evaluate the calculated information in measurement class. These objects read from the database in the beginning of system start-up and get a variable. This variable can be of any type and contain any information to be used for comparing data in measurement objects against. In the simplest case it can be a simple integer which is a limit and the measurement object data may not go higher than this value. It can also be more advanced to represent e.g. a percentage for the evaluator object to use in its evaluation method.

If the evaluator object comes to a conclusion that the limit has been exceeded, it will save current time and measurement data in a list of alarms. The triggered alarms list is a circular list and saves a pre-defined amount of last events per evaluator object.

The whole framework is generic and so is the triggered alarm list. The base class only contains a timestamp and functions for set and print. The developer needs to inherit from the base class and create the variables needed for saving the important information about the specific alarm. A triggered alarms time class was created for this specific purpose and will serve all functionality needed when creating an alarm entry when a time limit was exceeded.

The purpose was to create a base class that only contains the most important functions and data. A time stamp will most likely always be interesting, no matter what kind of measurement and/or evaluator the developers continue with. The time alarm class contains a variable for saving the measured time data at the exact moment the limit was exceeded.

The developer will have the possibility to see all triggered alarms by printing them in a terminal. When printing alarms the developer gets information regarding what manager and measurement the alarm belongs to, what the limit for the alarm was, at what time the alarm happened and what data lead to the alarm. This information will help the user when debugging a system by giving information on when a system entered a state which gives errors.

5.6.2 Lifecycle handling

No deletion of objects is done directly by a developer; all deletion is done with the reference counting, e.g. when an object referencing to the manager stops using it, it just does a decrease of the reference count. By doing this, the manager knows that this object has no interest in it any more. When the number of references to the manager is zero, the manager is not needed any more and could be removed. When this happens, all measurements connected to the manager are deleted.

The probes must not be deleted; it is enough that the manager decreases the reference count to them. This is because there might be a pointer cached somewhere in the code that stamps it. The deletion is done by returning how many references the probe have each time it gets stamped. By returning it, the code snippet will check if the reference count is one or higher. If it is just one, the probe reference count can be decreased and the pointer deleted since no one but the stamping section itself is interested in it

System deleting is of a slightly lazy type. Probes could be removed as fast as no measurement is interested in them, but the decision was made that it is better to have a bit more memory usage and deleting them less often instead of frequent deletion and recreation of probes, which takes more CPU time.

5.6.3 Communication and component outputs

Alarms in the system get their configuration from the database and their limits and parameters are therefore easily changed by e.g. reading an XML file into the system. It can also be of use to be able to read different measurements and results in other components, e.g. the HMI or in a separate test environment. As explained earlier, outputs from components can be used for this purpose. Presenting measurement data, alarm status and other important parameters can be great when monitoring the system either by hand or by an automatic test environment.

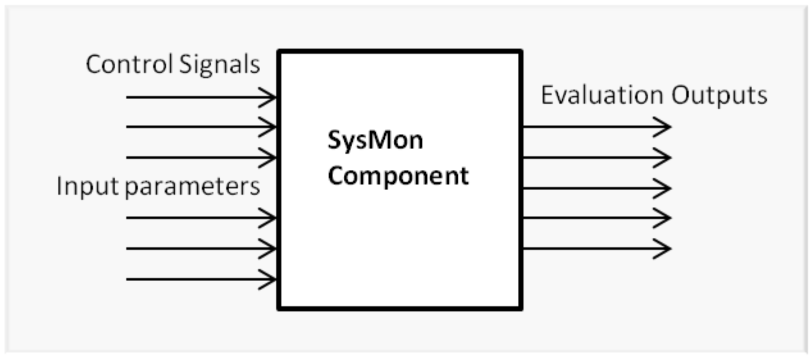


Figure 10: Inputs and outputs to a SysMon Component

Runtime objects already have their own component for measuring purposes. These components will be used for handling the input and outputs for its measurement manager.

There are several places of the code that is interested to be measured that do not live inside a runtime component. One good example is communication protocols. To be able to get output from these measurements a new component is created for connecting to the measurement manager of measurement that lives outside runtime components. This component will run within runtime in a thread and get its data from a measurement manager, and make this data visible on its outputs. Each measurement manager needs their own component so a 1:1 relation of component instances and measurement managers outside runtime is needed.

5.6.4 Version handling

To reduce the amount of double work in the tool, a system for version handling has been implemented. The probe- and measurement-classes has got a version number associated with them, which is a counter.

Each time a probe gets stamped, the version number for it is incremented. When a measurement calculation is activated, it checks the version of both probes and if both versions have increased since the last time, the calculation is done. The versioning in probes gives two positive aspects for the system. Calculations for probes are always just done once for every version, and the versioning guarantees that the measurement is consistent in the case that two probe values is compared. Consider a scenario where just probe 1 has been updated while probe 2 has not, and these are compared. The result will be incorrect.

To reduce the number of evaluations, the measurements also have version handling. Each new complete measurement has a new version and when evaluations are run, the version for the measurement is checked to see if it should be evaluated or not.

5.7 Class description

From the conceptual model the methods and variables for the different classes has been constructed. The designed system is built up by a number of main classes which is of great importance for the system behavior. This section describes the most important classes with respect to the functionality and conceptual methods.

5.7.1 Probe

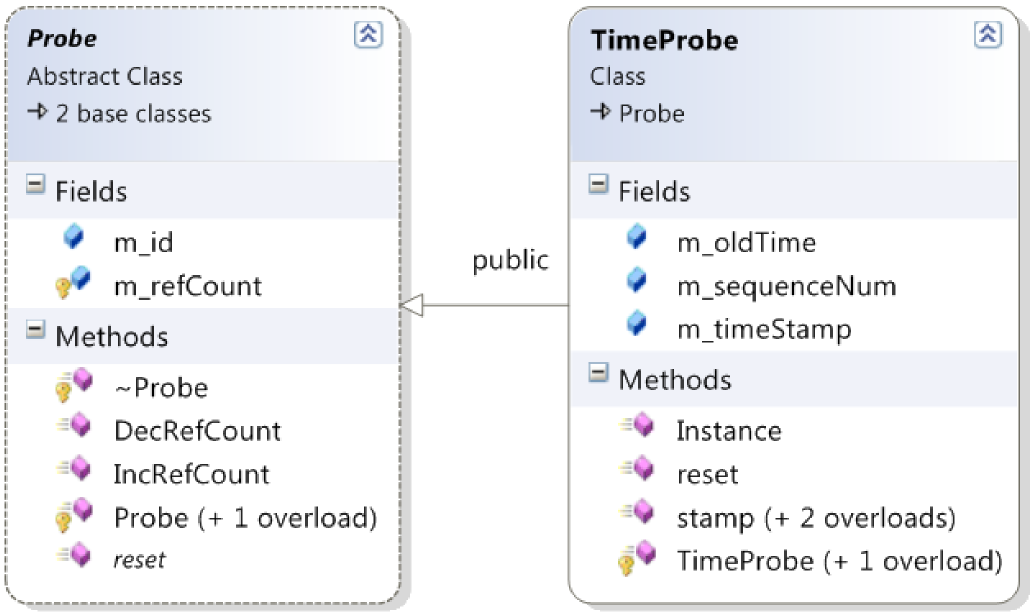


Figure 11: Probe base class and time probe implementation

The probe-class is used to create probing-objects that are assigned data from the specific points of the code. The information the probes take can be of various types, like a timestamp, size of a buffer, number of invocations of a task etc. To make it possible for the numerous of different usage areas, a virtual probe-class with the basic functionality is created which then is used with inheritance to make new classes for the specific probe.

All probes will have a name that is unique within the manager it belongs to. With this unique name, everyone that has a reference to the manager can find it. An OID, object identifier, is used for the naming. The naming convention is a question for later system design but the names of the probes must make sense in a way that it is easy to know and find a probe when a measurement is created.

A probe is instantiated when the need for it exists, e.g. when a measurement needs the probe for its measurements. Each probe has a variable that counts the number of references to the object. When a new measurement is created and associated to the probe-object it does an increase of the reference counter. When the number of references is zero the probe is not used in any measurement or by any time stamping in the system and therefore removed.

To save execution time and to minimize the need of searching in lists, stamping of a probe is done directly with a pointer to the stamp. Each probe has a stamp-method that is used to take a stamp from the system. By calling the stamp-method with the data to be stamped, the data is saved in the probe. For e.g. timestamps there are methods that take no in parameter and the meaning is to save the current timestamp as data.

5.7.2 Measurement

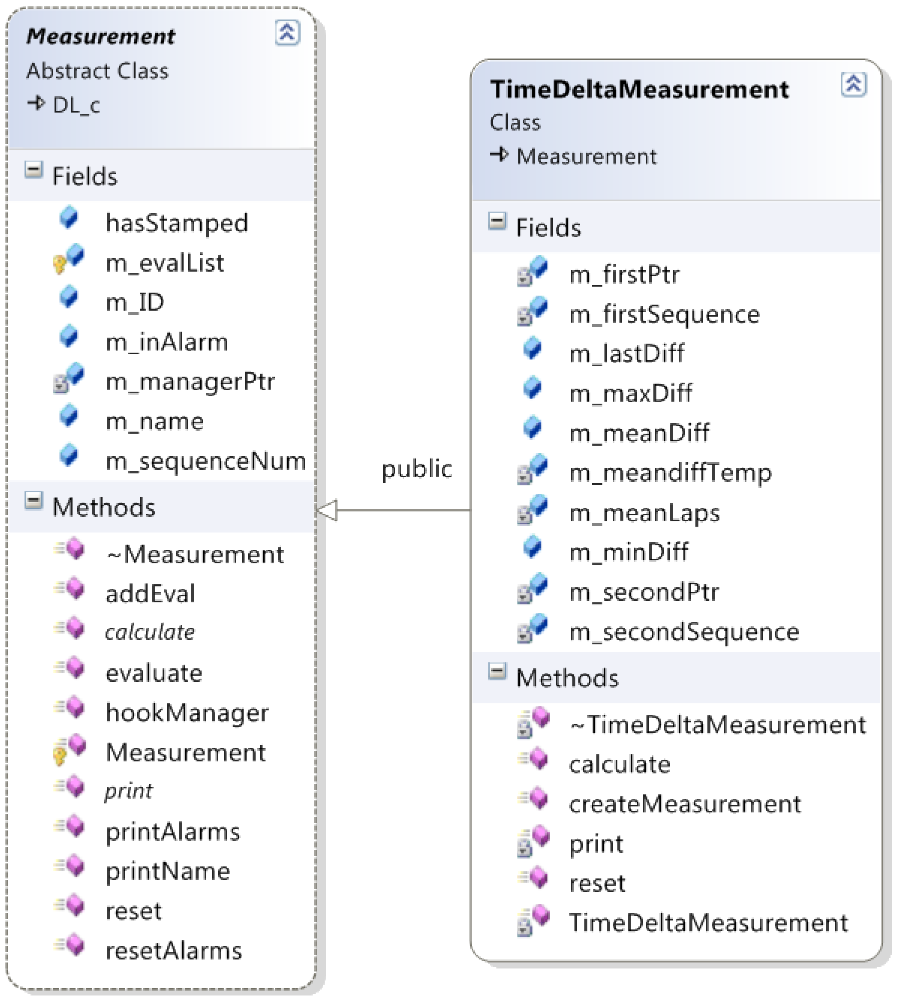


Figure 12: Measurement base class and specific implementation

The measurement uses probing information to calculate specified measurements for the system. Figure 12 shows the base class of measurement and an implementation for a measurement that calculates time differences between two time-probes.

The measurements base class has a name of the measurement, an integer ID, and a list of evaluations associated with the measurement. It has virtual methods for printing and calculating the measurement, which inherited classes' implements, and also an evaluation-method which calls all the evaluators in the measurement to do its evaluations.

The specific implementation shown, a measurement for time differences is used for comparing the time between two probes in the system.

5.7.3 Measurement Evaluation

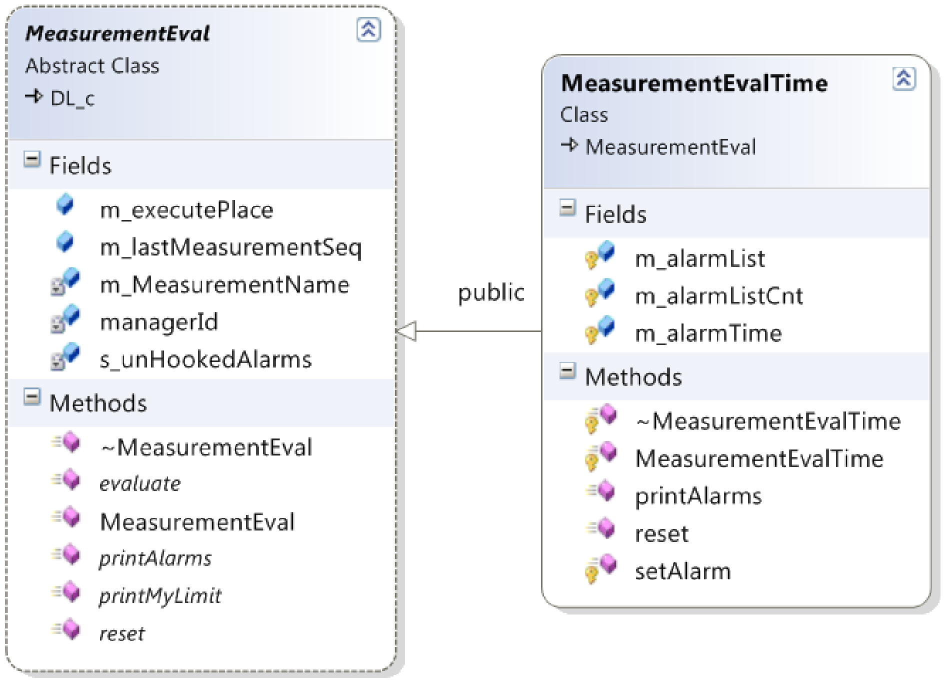


Figure 13: Measurement evaluation base class and time evaluation implementation

The evaluation class is the one containing the “intelligence” of the system. While probes and measurements just gather data and do simple calculations like calculating minimum and maximum value, the evaluator processes the data and takes decisions based on the information received from measurements. This could be that it compares jitter of executions, maximum execution times with deadlines or similar. If certain values of the result as achieved, the evaluator could be configured to execute certain code. This could be to stop a recorder and save the current state or more advanced functionality like controlling the actual execution flow of the system, e.g. giving the executing task less CPU time slots because of too long execution time.

The base class of the evaluator contains a list of triggered alarms and a variable that tells the system where to run the evaluator. The triggered alarms list is a circular buffer which saves the time for a pre-defined amount of alarms and what data that the alarm is associated with. This could help a developer debugging by showing at what exact time an alarm occurred.

The other variable, the execution place, is used to decide on where to run the evaluator. The thought is to have a possibility to configure where the tool evaluates measurements in the future. Either the evaluator can run directly at a measurement update, it can run when the user enters the evalAlarms-method or in a separate thread. Where to run the evaluator depends on the responsiveness required and the amount of load an evaluation does. If the evaluator just does simple calculations like comparing values, it can run in context of the evalAlarms-method, but if it gets too complicated and time consuming a good idea could be to put the evaluation in the separate thread. When this is done the evaluation is activated by a lower priority task that does less interference with the rest of the system. A remark is that the developer must take the time it can pass from an actual system error until it is evaluated and discovered when running anywhere else than in the current context into consideration. This is because the trace-recorder does not save many seconds of history when logging in the ring-buffer. The evaluation should be run so fast after the error as possible to get a good recording of how the system got into the error state.

The MeasurementEvaluation class also has a debug symbol built into the system standard concept of debug symbols. By turning it on, the developer can see live outputs when an alarm is triggered. Additional information is also printed for identification of which alarm was triggered and what limit it exceeded.

Figure 10 also shows a specific implementation for a time evaluator. This is a simple example and only contains a hard deadline time with the variable m_alarmTime. At evaluation this value is compared to the measurements maximum probing time since the last alarm and, if the time is no within defined limits, an alarm is produced. Examples of our specific inherited classes are:

- **MeasurementEvalTimeHighLimit** - *Checks if a calculated time is higher than the set alarm time*
- **MeasurementEvalTimeLowLimit -** *Checks if a calculated time is lower than the set alarm time*
- **MeasurementEvalTimeRelativeLimit -** *Checks if a calculated time differs in a set percentage versus the mean time*
- **MeasurementEvalTicksHighLimit –** *Checks if a calculated amount of ticks is higher than the set alarm*
- **MeasurementEvalTicksLowLimit –** *Checks if a calculated amount of ticks is lowwer than the set alarm*
- **MeasurementEvalTicksRelativeLimit –** *Checks if a calculated amount of ticks differs in a set percentage versus the mean time*

5.7.4 Triggered alarms

Triggered alarms are used to record and save information about evaluations of measurements that has given a result that is breaking the evaluation limits. When an evaluator gets an alarm state, the alarm data is saved in a circular buffer of triggered alarms. The information stored in the buffer is the system time when the alarm happened and the data which lead to the alarm.

5.7.5 Manager

The manager is the key object in the developed system. The manager is responsible of keeping track of the measurements and probes associated to it and it is used to get pointers to existing probes. It is with the help of managers that it is possible to connect different threads and parts of the code to the same probes and measurements.

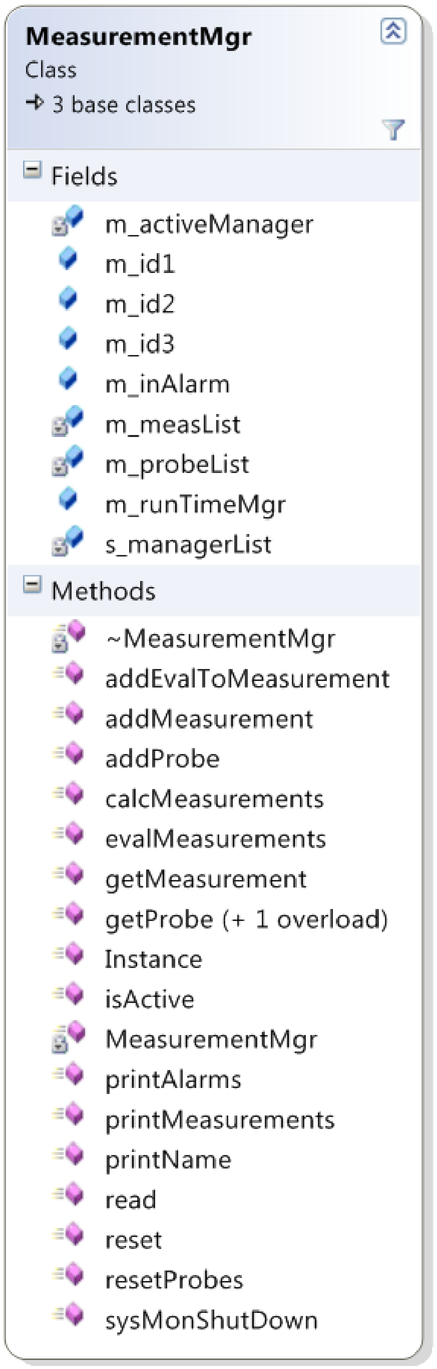


Figure 14: Manager Class

All initiation is done with the static instance-method in the manager class. With the help of the static manager-list, the method searches through the existing managers to see if one already exists with the specified name.

There are a number of important variables in the manager where the two most important is the list of measurements and list of probes.

5.8 Using SysMon

Most of the communication between system components and the monitoring is done via the manager; the only exception is when stamping probes, which is done by addressing the probe directly with a pointer. This is because each lookup in the probe list costs execution time and a pointer is the least expensive way to assign values to variables.

To give a better idea on how the framework is thought of using in a real system some examples are given in this section. This example uses no external source for the alarm; instead the alarm-object is created with a variable that exists in the object creating the measurement.

5.8.1 Initializing SysMon manager and measurements

The first step when construction a new monitoring object is to set up the manager. As manager can be used on multiple places, an instance-method is used to find an existing manager or create a new manager if none exists.

```
m_managerPtr = MeasurementMgr::Instance(managerName);
```

The next step is to create the probes needed for setting up the wanted measurements

```
m_wakeProbePtr = TimeProbe::Instance(m_managerPtr, wakeProbeName);
m_doneProbePtr = TimeProbe::Instance(m_managerPtr, doneProbeName);
```

When probes for a measurement are created, the measurement can be created and added to the correct manager. As parameters to the create method is a pointer to the manager, index for measurement, name for measurement and pointers to the probes that should be used by the measurement.

```
TimeDeltaMeasurement::createMeasurement(m_managerPtr,0,"W2D",  m_wakeProbePtr,
m_doneProbePtr );
```

If wanted, add an evaluation to the created measurement. As parameter to the create evaluation method is a manager pointer, index of measurement and an alarm value for the evaluator.

```
MeasurementEvalTimeHighLimit::createMeasurementEval(m_managerPtr,0,
m_alarmList[0]);
```

The framework is now configured for doing the measurements. The next step is to start the monitoring.

5.8.2 Setting up probe points and doing calculations and evaluations

To set up probe points for taking stamps of the system, a reference to the wanted probe must be fetched from the manager. If the stamping is done in the same object as the initialization, a reference is already present from the previous step. The false-flag tells the instance-method that it shouldn't create the probe if it doesn't exist. As this object is only responsible of stamping a probe it is not interesting to do so if no measurement wants to use it in a measurement, therefore the flag is set to false.

```
m_wakeProbePtr = TimeProbe::Instance(m_managerPtr, wakeProbeName, false);
```

To stamp the probe, add a stamp-method call at the position in the code where it is suitable.

```
m_wakeProbePtr ->stamp();
```

When all information is gathered for the measurements, run the calculation in the manager to do calculation on data from probes.

```
m_managerPtr->calcMeasurements();
```

Next step is to evaluate the calculated result from previous method by running the evaluation method.

```
m_managerPtr->evalMeasurements();
```

6 Testing

The tests done on the tool are in two main areas. First the tool itself is tested to see that it works as planned. This means both that the tool doesn't produce any bugs or error and also that the values measured are reasonable. The second part of testing is to do actual measurements on the company system to see if the tool runs stable, produces correct measurements and lives up to the high standards of ABB's system.

6.1 Test lab environment

ABB have a huge testing area for every kind of scenario the system might encounter. In the lab rooms there are cabinets which can freeze the system down to under -50°C and also heat it up to over +50°C. There are also all kinds of testing equipment for simulating voltage and power as well as communication. A few people at the company only work with test scenarios and developing tests for the system that might happen in the natural environment and should be handled without problems. The binaries for the system are built every night using the newly checked in code from the developers and after every build there is an automatic test process. The test process runs a thorough test program and gives feedback to the developers if it passed all tests and gives information if a test did not pass.

The testing is an important part of the development of the product. It is also an important factor to recreate problems that have occurred in normal situations and then reported as failures from the customers. Sometimes there is little information about the actual failure and how it occurred. Recreating the same situation in a lab environment is therefore sometimes a tough job. But it is extremely important to try to understand what happened and how it can be prevented since an unnecessary power cut is not acceptable.

6.2 SysMon test process

The test process has three parts, which is run in an iterative way during the development and testing process. Each of the tests has its positive aspects and helps the development by showing bugs and errors and giving results from measurements.

The first step in the test process is to run the executable in Windows environment. This environment does not function exactly as the target system, but is always a good first step in testing new code. Running the code and being able to set real breakpoints in Visual Studio is a lot easier than compiling and running straight on target system without being able to debug more than writing error dumps upon failure.

SysMon was tested a lot in Windows environment, and even though times and real-time properties does not function well in Windows, many bugs and errors in the design were discovered in this stage.

The next step in the test process is to compile and run the executable in target environment. Since VxWorks and Windows handles things different and a few code snippets of the system are different whether it is running in Windows or VxWorks this can reveal bugs and unexpected features. This is a important step, and running newly written code in Windows only does not prove that it works.

The final step of testing is to run the tool in lab environment on a test setup where the system gets actual inputs and outputs from external hardware. This gives a good picture on how the system behaves at a real install and is a good last step for testing SysMon.

6.3 Tool evaluation and benchmarking

The first step of evaluating the tool is to expose it to different execution scenarios. It is important for the tool to handle all special system specific situations. The tests were designed to cover the most critical and important system execution scenarios including warm reboots, normal lifecycle of communication links and long time tests. An important test is also to test the system with SysMon disabled by a parameter. By setting SysMon enable parameter to false, the system should function as normal and SysMon should not run any of its code and therefore not affect the system in any way.

Another good test is to simulate load on the system to force alarms to trigger. This can be done by inserting an extra task that has a simple dummy spin loop and runs with a period so that it will affect a pre-defined normal system task. Another way to trigger alarms would be to simply lower the limits and making the alarm limits get exceeded each execution, but it is more realistic to simulate load.

The evaluation tests are executed on a local system with no inputs or outputs. Descriptions and results of the tests can be reviewed in Table 3.

Benchmarking can be described as tests that evaluate the effectiveness and intrusiveness of SysMon. An important factor to test is the CPU load of the probing of the system in relation to the probe effect described earlier in the thesis. The measuring method used was proposed by Kraft [20]. The first step is to evaluate how long time an actual probing in the system takes. To measure this, two probes were added in the system that measured system ticks that it takes to do an actual probing. The probes were put right after each other, and the difference in time will then be the time it takes for a real probe to execute.

Calculation and evaluation functions are also important to evaluate since they contain the mathematic and CPU intense parts of SysMon. By adding two probes to take time stamp before and after calculation and the same for evaluation functions, the intrusiveness of these was measured. The scaling of the tool, e.g. how the tool behaves when adding more than one evaluator objects to an already existing measurement is interesting since measuring just one evaluator can have code that only executes once, e.g. locking of shared resources. Scaling tests are done with different amount of measurements, evaluators and alarms that triggers.

A general CPU load test was setup with a standard execution scenario. Ten application threads were probed four times during one execution cycle and these four probes were used by six time measurements. Each measurement also had one evaluator that did not trigger an alarm. It is important to mention that these threads are not the only threads in the system, but they are the most interesting to monitor at the moment. The threads monitored have a schedule scenario as presented in Table 2.

Thread number	Periodicity (s)	Executions per second
1	1	1
2	1	1
3	1	1
4	0,1	10
5	1	1
6	0,005	200
7	0,02	50
8	1	1
9	0,2	5
10	1	1

Table 2: Thread periods in CPU load analysis

The tests were executed both on a local system with no input or input signals and also one time during a stress test of the system. The stress test is developed at the test department and is a standard test for evaluating CPU load.

6.4 System test

A number of predefined system tests are available to execute in the lab environment. These will be used to measure and evaluate specific system properties and how the system performs under different execution scenarios. This is not a test for the tool itself, but for the system. The exact test specifications and results from these measurements will not be presented in this report.

6.5 Test results

A number of test cases have been constructed to test the parts described in the previous sections. The description of the test cases and the results when running them are presented below. Specific tests were also done on the company system for evaluating the target system and its properties. These test results are only available internally on the company for review.

6.5.1 Tool Evaluation

ID	Test description	Expected result	Result
T-01	Run system with SysMon disabled	Measurements do not get updated and no evaluation is allowed to run.	PASS
T-02	Disabling and enabling SysMon during runtime	On disable measurements and evaluation values remain intact and when enabled again, they get updated with valid data	PASS
T-03	Disabling and enabling system runtime while SysMon is enabled	Measurement data of runtime objects does not get updated when runtime is down, and then starts to update with correct values once runtime is enabled again	PASS
T-04	Communication links is enabled and disabled while SysMon is enabled	Measurement data does not get updated for the specific link measurements, and then starts to update with correct values once enabled again	PASS
T-05	Running system for 24h nonstop with SysMon enabled	All measurement gets valid data and no exception occurs	PASS
T-06	Simulate load to trigger an alarm	Alarm gets triggered, saves correct data and sets correct output values for the component	PASS

Table 3: Tool evaluation results

6.5.2 Benchmarking

ID	Benchmark description	Result
B-01	Measure system ticks for one probing event	Mean ticks: 12 (~0.18 µs) Min ticks: 6 (~0.09 µs) Max ticks: 39 (~0.59 µs)
B-02	Measure system ticks for one lap of measurement calculations – 1 measurement with updated values that will run its calculations	Mean ticks: 86 (~1.3 µs) Min ticks: 50(~0.76 µs) Max ticks: 205(~3.1 µs)
B-03	Measure system ticks for one lap of measurement calculations – 10 measurement with updated values that will run its calculations	Mean ticks: 714 (~10.83 µs) Min ticks: 591(~9.00 µs) Max ticks: 1012(~15.33 µs)
B-04	Measure system ticks for one lap of measurement evaluations –1 evaluator, 0 alarm that triggers	Mean ticks: 122 (~1.85 µs) Min ticks: 86(~1.30 µs) Max ticks: 200(~3.03 µs)
B-05	Measure system ticks for one lap of measurement evaluations – 1 evaluator, 1 alarm that triggers	Mean ticks: 217(~3.29 µs) Min ticks: 106(~1.60 µs) Max ticks: 330(~5.00 µs)
B-06	Measure system ticks for one lap of measurement evaluations – 10 evaluators, 0 alarm that triggers	Mean ticks: 392 (~5.94 µs) Min ticks: 289(~4.38 µs) Max ticks: 467(~7.08 µs)
B-07	Measure system ticks for one lap of measurement evaluations – 10 evaluator, 10 alarm that triggers	Mean ticks: 1065 (~16.16 µs) Min ticks: 392(~5.94 µs) Max ticks: 1153(~17.47 µs)
B-08	Average CPU load increase with SysMon enabled	Avg. CPU load SysMon disabled: 14.0% Avg. CPU load SysMon enabled: 14.64% See Figure 15 for CPU Load diagram
B-09	Average CPU load during stress test with SysMon enabled	Avg. CPU load SysMon disabled: 45.53% Avg. CPU load SysMon enabled: 46.66% See Figure 16 for CPU Load diagram

Table 4: Benchmark results

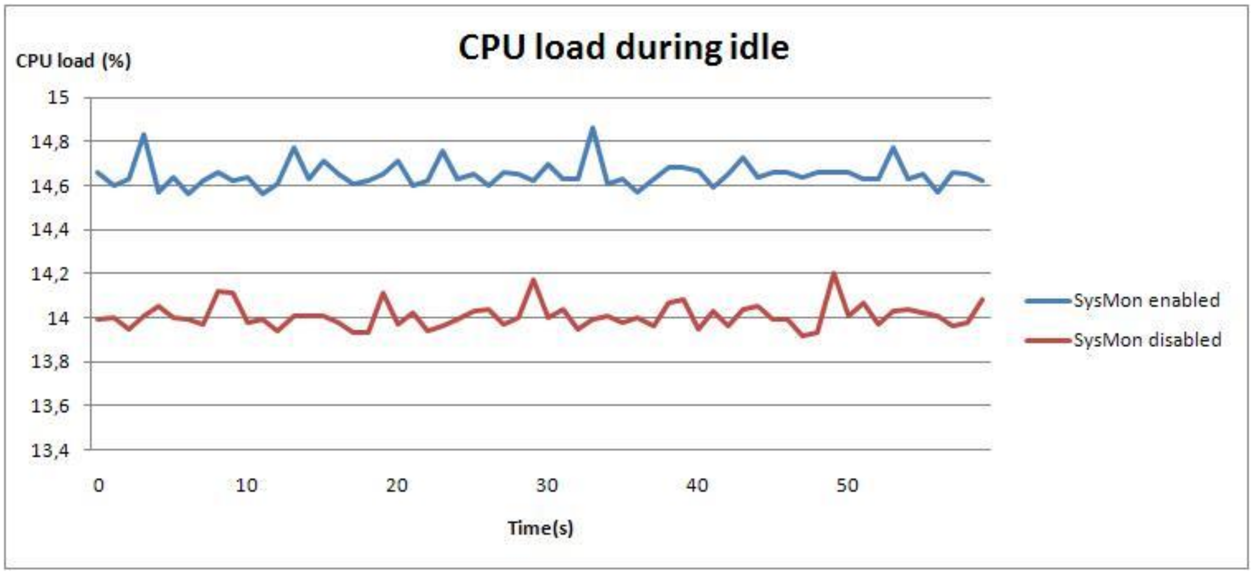


Figure 15: CPU load during idle

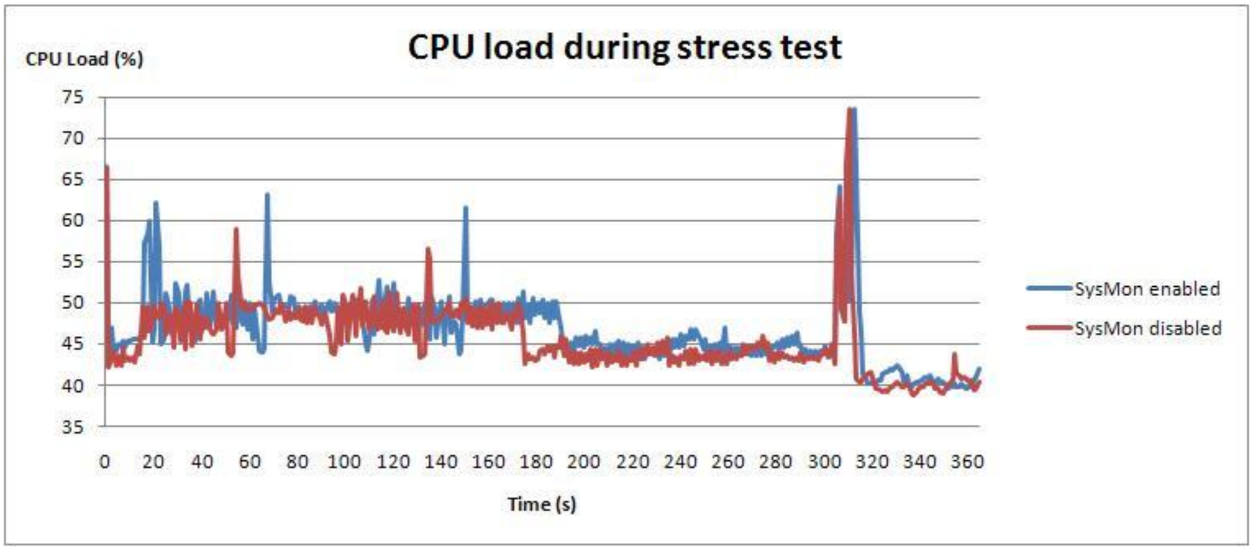


Figure 16: CPU load during stress test

6.6 Test discussion

Given the results from the tests, what conclusions of the tool can be drawn? First of all, regarding the evaluation, the tool runs stable during the scenarios tested. These scenarios should be enough to find critical errors like memory leaks and pointer errors. It also shows that the system runs stable during a fairly long execution time, around 72 hours. Even though these tests, one can still discuss if the testing is thorough enough to guarantee a stable run. The ABB system observed is designed to run for extremely long times without interrupts and tests with these long testing times, weeks or maybe even years, is not possible within the time frame of the thesis work. This shouldn't be a problem though as the system runs in a cyclic matter and with our one day-test all threads has run for a lot of cycles.

Another problem with the tests is that they don't test the system with all different types of stimuli, but considering the design of the tool and the type of usage, different stimuli shouldn't give different results for our tool as the tool execution pattern doesn't change at all with different execution traces of the system. The monitored threads will still have the same periodicity; the only difference is that they will have more work to do during the execution. The tool only gathers information for probes, does measurements on them and evaluates the measurement, no matter what happens in the rest of the system.

To have a possibility to use the tool on an effective and useful way, it must not only do what it is supposed to do but also in a fast way with a low CPU load on the system. Therefore the benchmarking tests are of high interest. The ticks and times measured for different usage of the system gives a good picture on the impact of the system. All times measured must be seen as approximation and not real facts, they should be used to give an idea in what range the monitoring code takes. This is because the system can sometimes interrupt the measured code and therefore give unrealistic values for the measurements. When calculating maximum and mean values, the results that were unrealistically large were not taken into account for the results presentation. These were values in the range of 4-10 times larger than any other values, and as the code executed were the same for all monitored threads, this should not be possible to get without interruption.

The CPU load test is maybe the most important test as this gives a good picture of the average CPU load increase in a scenario that should be fairly standard when using SysMon, regarding the number of probes, measurements and evaluation. What the result of this shows is that the load increase of the system is in fractions of a percent, the difference between the lowest measured load without SysMon and the highest measured with SysMon is only 0.94 %. This CPU load increase should be small enough to make the enabling of SysMon possible in system tests without interrupting the normal execution pattern in the system in a way that affects the system interactions.

The CPU load during stress test shows about the same numbers, the average CPU increase is around 1.1%. Figure 16 shows CPU load during a stress test, and many parameters affect the current CPU load. One can see that the figure sometimes show less CPU load with SysMon enabled than disabled. This could be because the tests were run one at a time in series, and the curves may not be exactly synced in time. The CPU load is also measured as an average per second and reported back to the developer once every second. Some variance between where the cutoff for a second is can affect the reported CPU load, since the test software stimulates the system with different signals throughout the test. Peaks and other abnormalities in the diagram which do not reflect in both curves are probably coincidences, and may differ because of the system itself and most likely have nothing to do with SysMon. The most important result from the test is the average CPU load over time, which shows an increase of acceptable level.

7 Conclusion

It has been an extremely interesting and fun thesis with a lot of mixed experiences. The plan has changed several times, from integrating an existing tool, to developing a new.

The target system is huge and it took a lot of time to understand the system, how it is built up and how the previous measuring tool, JobMon, was designed. It also took a bit of time to understand what kind of tool the company actually wanted and features were added until the very end of this thesis time.

The outcome of this thesis is a generic framework that can be used for measuring any type of data within a system. Measurements can be evaluated against limits set by parameters in a database and take desired action depending on the result.

The framework has been tested thoroughly during the development and two totally different types of measurements have been implemented; time and ticks. Even if they both measure time in their meaning, the data types are different and show that the framework can handle different types of data without problems.

The possibility to construct measurements with our framework as a base is almost endless. No restrictions have been made for what kind of data it can measure and the developers at the company can continue develop this software to exactly what they need in the future.

The tool can run forever in the system since all design is based on ring buffers and the monitoring does not infer the system in a noticeable way. Tests show that the tool does not take noticeable amount of resources and therefore should not affect other parts of the system. It provides a simple concept of saying that a limit e.g. a time limit for a task has not been exceeded since the tool started measuring.

These types of measurements are extremely important in products that have a huge demand in uptime; time without error. The normal commercial debug-utilities presented earlier in this thesis show a short trace log. They do not help much if you have one error per month, or even one error per year. Our tool provides the features of finding and locating the exact moment of an error. It can measure, evaluate and upon failure save logs and/or trigger other utilities at a minimum amount of time after an error.

8 Future of SysMon

As the SysMon implementation done in this thesis is just a first prototype, a number of improvements can be done to make the tool work better. This thesis had a limited amount of time, and even if it would be good to implement all functions suggested by ABB and the authors ideas it was necessary to cut the work somewhere.

First of all, more testing and evaluation should be done on the tool to further guarantee its correctness. The system is designed to run during long sessions and the tool must produce correct results during long-time testing and remain stable during the whole execution time.

The communication to the tool with inputs, outputs and parameters could be improved to make it possible to control more of the tool functionality from en external source. Right now the control functionality from external sources is more or less limited to just resetting calculations, do a complete disable of manager(s) and setting alarm values. More advanced pausing/resuming functionality and other control signals could be added in future improvements to make it possible to configure the tool more individually.

Another improvement that was taken up during the thesis is to integrate another system recorder/system analyzer together with SysMon, that records information about the system like task switches, semaphores and so on. With the help of SysMon evaluations system properties could be set up and if these are broken the recorder could be stopped and the log examined to give further information regarding errors in the system. This improvement is more about configuring the external tool to fit the needs of the developer, more than configuring our tool. Our tool just needs the correct function syntax for triggering the external tool at the correct place since everything is prepared for this scenario.

Regarding the CPU load of the tool, some improvement should be possible to do in that aspect as well. If the tool is considered too heavy for the system the code could be examined to find time-heavy parts and, if possible, make the code faster.

9 References

[1] ABB. (2012, February) Kraftnätets väktare. [Online]. http://www.abb.se/cawp/seabb361/2d15bfc06420a186c12572510030282c.aspx

[2] Werner Schütz, "On the Testability of Distributed Real-Time Systems," in *Proceedings of the 10th Symposium on Reliable Distributed Systems, Pisa, Italy*, 1991.

[3] Hermann Kopetz, "Should Responsive Systems be Event-Triggered or Time-Triggered," Institut für Technische Informatik, Wien, Research Report Nr. 16/93 1993.

[4] Hermann Kopetz, "The design of fault-tolerant real-time systems," in *EUROMICRO 94. System Architecture and Integration. Proceedings of the 20th EUROMICRO Conference*, Wien, 1994, pp. 4-9.

[5] Hermann Kopetz et al, "The design of large real-time systems: the time-triggered approach," in *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, 1995, pp. 182-187.

[6] Jukka Mäki-Turja, Kaj Hänningen, and Mikael Nolin, "Efficient Development of Real-Time Systems Using Hybrid Scheduling," in *International conference on Embedded Systems and Applications (ESA)*., June 2005.

[7] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, 1973.

[8] Yoshifumi Manabe and Shigemi Aoyagi, "A Feasibility Decision Algorithm for Rate Monotonic and Deadline Monotonic Scheduling," *Real-Time Systems*, vol. 14, no. 2, pp. 171-181, 1998.

[9] N.C. Audsley, "Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times," Dept. of Computer Science, University of York, England, November 1991.

[10] Dorin Maxim, Oliver Buffet, Luca Santinelli, Cucu-Grosjean Liliana, and Rob Davis, "On the Optimality of Priority Assignment for Probabilistic Real-Time Systems," in *19th International Conference on Real-Time and Network Systems - RTNS 2011*, Nantes, France, Nantes, France, 2011.

[11] Johan Andersson, "Modeling the Temporal Behavior of Complex Embedded Systems," June 2005.

[12] L. Sha et al., "Real Time Scheduling Theory: A Historical Perspective," *Real-Time Systems*, vol. 28, no. 2/3, pp. 101-155, 2004.

[13] J.A. Stankovic, M. Spuri, and M. Di Natale, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, pp. 16-25, June 1995.

[14] N.C. Audsley et al, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling,"

Software Engineering Journal, vol. 8, no. 5, 1993.

[15] Andrew S. Tanenbaum, *Modern Operating Systems*, 2nd ed.: Prentice Hall, 2007.

[16] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed., 2005.

[17] Reinhard Wilhelm et al, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, April 2008.

[18] Jan Gustavsson, "Usability Aspects of WCET Analysis," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, May 2008, pp. 346-352.

[19] Henrik Thane and Hans Hansson, "Using Deterministic Replay for Debugging of Distributed Real-Time Systems," in *12th EUROMICRO CONFERENCE ON REAL-TIME SYSTEMS*.: IEEE Computer Society, 2000.

[20] J Kraft, "Enabling Timing Analysis of Complex Embedded Software Systems," August 2010.

[21] Percepio AB. (2012, May) Percepio. [Online]. http://www.percepio.se/

[22] Express Logic. (2012, February) TraceX. [Online]. http://rtos.com/products/tracex/

[23] WindRiver, *System Viewer User's Guide, 3.0.*, 2007.