FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Time-predictable Operating System

Towards a Constant Execution Time

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Christopher Helpa
Matrikelnummer 0226438

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn.Peter Puschner

Wien, 31. August 2011

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

A Time-predictable Operating System

Towards a Constant Execution Time

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Christopher Helpa
Registration Number 0226438

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn.Peter Puschner

Vienna, 31. August 2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Christopher Helpa
Schumanngasse 32/10, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Real-time systems are used in a wide range of applications ranging from traffic light control to spacecraft navigation. The correctness of a real-time system's action does not solely depend on the logical correctness of its computations, but also on the instant of time at which the results are available. Results that are not available within a certain amount of time can result in catastrophic events. To prevent this an integral part of system development is determining the worst case response time of an application.

Providing operating system mechanisms that ensure temporal isolation of all tasks makes it easier to determine the worst case response time of a task. Calculating the worst case execution time (WCET) of an application is one crucial step in determining the response time of a system. Unfortunately, as a result of the complexity of modern systems, calculating the WCET can still require a huge effort.

This thesis demonstrates the development of a real-time operating system that ensures temporal isolation of all tasks, as well as constant execution times of all its functions. Subsequently, analysing the execution time can be carried out by a small number of measurements.

The programming approaches and operating system concepts crucial to these goals are discussed, and the details of the implementation as well as insights gained from the implementational experience are presented.

The evaluation showed that a small number of measurements is sufficient to determine all function's temporal properties, as well as affirmed that the execution times of all operating system functions are constant.

Kurzfassung

Die Anwendung von Echtzeitsystemen reicht von der Steuerung von Ampeln bis zur selbstständigen Navigation von Raumschiffen. Bei Echtzeitsystemen hängt die Korrektheit des Systemverhaltens nicht nur von der logischen Korrektheit aller Berechnungen sondern auch von dem Zeitpunkt zu welcher die Resultate vorliegen ab. Resultate die nicht innerhalb einer gewissen Zeitspanne vorliegen können katastrophale Auswirkungen haben. Um dies zu verhindern ist die Bestimmung der maximalen Reaktionszeit eines Programms von entscheidener Bedeutung.

Betriebssystemmechanismen, welche die zeitliche Isolation aller Anwendungen garantieren, vereinfachen die zeitliche Analyse von Anwendungen. Die Bestimmung der maximalen Ausführungszeit einer Anwendung ist ein wichtiger Schritt in der zeitlichen Analyse eines Systems. Aufgrund der Komplexität moderner Systeme stellt die Bestimmung der maximalen Ausführungszeit leider immer noch einen hohen Aufwand dar.

Diese Diplomarbeit beschreibt die Entwicklung eines Betriebssystems das zeitliche Isolation aller Programme gewährleistet. Weiters weisen alle Betriebssystemfunktionen konstante Ausführungszeiten auf, welche durch eine geringe Anzahl an Messungen bestimmt werden können.

Im weiteren Verlauf werden die verwendeten Programmieransätze und Betriebssystemkonzepte vorgestellt, sowie auf die Implementierung und die dabei gewonnen Erkenntnisse eingegangen.

Die Evaluierung zeigt, dass eine geringe Anzahl an Messungen ausreicht um das zeitliche Verhalten aller Betriebssystemfunktionen zu bestimmen, sowie die Ausführungszeiten dieser konstant gehalten werden konnten.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Today computer systems are used in a many applications ranging from controlling traffic lights to navigating spacecrafts. Computer systems that are only part of a bigger system are called embedded systems. Often they have some control over object in the environment it is located in. Examples for such embedded systems are the flight computer of an aircraft or the motor control of a car.

These systems often have to fulfil strict timing requirements and are subsequently called real-time systems. Real-time systems differ from non real-time systems in that the correctness of calculation results also depends on the point in time at which the results are produced. The response of a system to an input has to be carried out within a certain amount of time to be useful. The point in time after which results become useless is called a deadline. Results that are not available before the deadline has passed have no use any more. In hard real-time systems missing a deadline constitutes a failure and can even lead to catastrophic events [Kop97].

As an example for such a real-time system, and to highlight the importance of predictable temporal behaviour, consider the release of an airbag during a car crash. It is not enough to ensure that the air bag opens but it has to open before the person hits the dashboard and is injured. The response time between the instruction to inflate the airbag until the point it is actually inflated must be smaller than the time until the person is injured.

To prevent such accidents it is essential to determine beforehand if the worst case response time of an application to an input is small enough to *always* meet the deadline. Timing behaviour is usually carried out in two steps. First the worst case execution time

(WCET) of all tasks in the system are calculated. Subsequently a schedulability analysis is performed to determine whether it is possible to execute the tasks in some order that allows all of them to meet their deadlines.

Systems where missing deadlines does not lead to catastrophic events, but still provide a useful service are called soft real-time systems. Examples are VoIP applications where not delivering a package in time might lower the sound quality but understanding the other person is still possible. In the following, when talking about real-time systems, we always mean hard real-time systems.

1.1 Motivation

Many features used by modern microcontrollers and processors are intended to optimize the common cases, owing to their big impact on responsiveness and hence usability of non real-time systems.

On the other hand these features often have no or even negative impact on the worst case execution time (WCET) and/or the temporal analysability of the application and need to be evaluated carefully before being used in a real-time system. Another problem is that the complexity of typical industrial applications tends to be too high for automatic timing evaluation due to the state explosion problem. Many methods require the assistance of an experienced engineer by annotating the source code with additional information about the program. Determining a safe upper bound on the WCET that way is an error prone and very time consuming task, which is in general only deemed economical for the most time sensitive and critical applications.

1.1.1 WCET-oriented Programming

One approach improving WCET analysis of applications for tools and engineers is to place the emphasis on algorithms and programming styles that have better temporal characteristics, like a better analysability or execution time jitter. Eliminating input data dependent control decisions is usually the first step required to make the temporal behaviour of applications possible to analyse. A more elaborate method that allows an application to consist only of a single execution path, from the start to the end of a code piece is the so-called single path approach. Using code transformations it is possible to generate code that consists of only a single path eliminating all needs for branches. This removes all the complexity that arises from the extremely high number of different paths potentially being traversed during the execution of the application. This eliminates the main obstacles in finding and analysing code for its WCET and makes the execution time constant in all cases. It is then possible to determine the (worst case) execution time of the program simply by taking a single measurement. The drawback of this approach is that it requires hardware support by providing predicated instructions which are not available on all processors [PB02].

1.1.2 Operating Systems

Due to the paramount importance of operating systems to software development it is essential to evaluate the suitability of operating system concepts for WCET-oriented and constant execution time oriented programming carefully [SR04]. Over the last decades real-time operating systems (RTOS) have been a heavily researched subject and many concepts have been developed that allow the execution times of operating system mechanisms to be bounded [SR04].

The requirement of only using algorithms whose execution time can be bounded is the main difference between real-time and traditional operating system. One basic idea, not only applicable to operating system code but all applications, is that if an upper bound on the execution time of every specific part of the system can be found, it is possible to determine whether or not the whole system adheres to the timing requirements. Evaluating small parts of the system instead of the whole system tends to make the temporal analysis easier by greatly lowering the number of states, required to be analysed.

A simple serialization of local worst case paths to form a global one usually provides a safe upper bound on the WCET but it could form a path which might not actually be reachable and therefore the real worst case path is shorter, leading to overestimation. This overestimation could result in the deployment of more powerful, than actually necessary, hardware, driving system cost up. Stricter requirements than merely having bounded execution times have to be applied to improve the temporal behaviour and analysis of a RTOS.

1.2 Goal of the Master's Thesis

This master thesis demonstrates the development of a RTOS that is time-predictable. This means that concepts specially tailored towards providing operating system functions with constant execution times are applied. Also good temporal analysability properties on the system level is required to determine the overall response time.

RTOS concepts have to be analysed for their suitability for reaching these goals and appropriate algorithms have to be designed and implemented.

The question we are trying to answer in this project is whether it is possible to write all operating system functions in a way that keeps the execution time constant. This is done by trying to eliminate all but a single execution path from all functions and equalize the execution time of the remaining paths if branches cannot be avoided. If it is not possible to have only a single path through each function, one would ask what additional measures have to be taken to reach a constant execution time and determine how large the jitter is in case it cannot be avoided.

The RTOS is to be implemented in the programming language C using the LEON3 soft core CPU by Gaisler Research because of its hardware characteristics, its extendibility and open nature. Another goal is to provide easy means to configure the operating system and to make it easy to port to other platforms.

The timing characteristics of the system calls are to be determined by measurements. A measurement approach is sufficiently suited for this purpose if the number of branches in the functions is sufficiently small. The test data is generated following a white box testing approach to determine the data that can have an impact on control decision.

The main goals can be summarized as:

- Evaluate operating system concepts for their temporal properties and derive methods suited for keeping the execution time of all operating system functions constant.
- These methods should furthermore be beneficial for the evaluation of the system's overall temporal behaviour.
- Programming methods suited for eliminating as many execution paths from the system as possible and keeping the execution time of remaining branches as constant as possible should be evaluated and applied.

- The execution times of the operating system functions have to be evaluated to determine whether execution time jitter exist.
- If execution time jitter persists it should be determined if it can be eliminated by changes to the code and evaluate the reasons if not.

1.3 Structure of the Master's Thesis

The rest of the thesis is organized as follows. Chapter 2 will give a short overview of WCET oriented programming and approaches that can guarantee a constant execution time. In Chapter 3 a brief introduction in RTOS concepts and related work is given. Chapter 4 will summarize the most important characteristics of the chosen hardware platform and software used, Chapter 6 will give an overview of the operating system and Chapter 7 subsequently covers the detailed implementation of said concepts and also covers how the operating system can be configured and extended. Chapter 8 will deal with the sample applications developed to demonstrate the operating system functionality and used for the measurement. The measurement results and lessons learned during the measurements and how the operating system was modified accordingly are also given. The thesis concludes with a short summary and some ideas for future work.

CHAPTER 2

WCET-oriented Programming

Introduction

This chapter starts with a short explanation of why WCET-oriented programming is needed. We talk about algorithmic concepts that are, respectively are not, well suited for WCET-oriented programming and list programming concepts that should be avoided. Methods used to improve the worst case path and the so-called single path approach are discussed subsequently.

2.1 The Goal

Traditional programming is concerned with providing high performance for the most frequent cases. By analysing the properties of the input data, certain methods are used to improve the performance for the most likely occurring cases. For example specialized functions for different data sets can be used instead of a single generic one to exploit certain data properties that can result in a speed-up. Also special checks for (trivial) corner cases can be performed. This leads to a shorter execution time in some cases.. Those, however, are not beneficial for real-time systems because they either harm analysability or increase the execution time jitter. Avoiding those is usually a good first step towards improving the WCET.

"WCET-oriented programming (i.e., programming that aims at generating code with a good WCET) tries to produce code that is free from input-data dependent control flow decisions or, if this cannot be completely achieved, restricts operations that are only executed for a subset of the input-data space to a minimum.“ [Pus03]

Additional goals Puschner states for WCET-oriented programming in [Pus05] are:

- Safe, simple, and high-quality WCET analysis
- Small execution-time jitter
- Good worst-case code execution times

2.2 Algorithmic Considerations

The first step towards good worst case behaviour is the selection of the appropriate algorithms with the same constant asymptotic complexity over the whole input space. If they have inherently different execution times depending on the input data they are not suited for real-time systems. As an example consider quicksort. On average and in the best case it has an execution time of $\mathcal{O}(n \log n)$ but in the worst case it requires $\mathcal{O}(n^2)$ time. Better alternatives for real-time systems are for example merge sort that is on average slower than quicksort, but always exhibits the same time requirement of $\mathcal{O}(n \log n)$ [Pus99]. Also often different algorithms are invoked depending on the data set size. Quicksort for example has a high overhead and for small data set sizes insertion sort is used instead that, although having a higher asymptotic complexity, is faster for small input sizes.

Considering the complexity of an algorithm alone is of course not sufficient because it does not consider additive and multiplicative constants at all.

2.3 Programming Optimizations Harming WCET

Many optimizations common to traditional programming have to be avoided for WCET-oriented programming as a result of their emphasis on optimizing the frequent cases but not necessarily the worst case. The idea behind this is summarized in Figure 2.2. By removing dependencies on the input data set the execution time is constant in all cases.

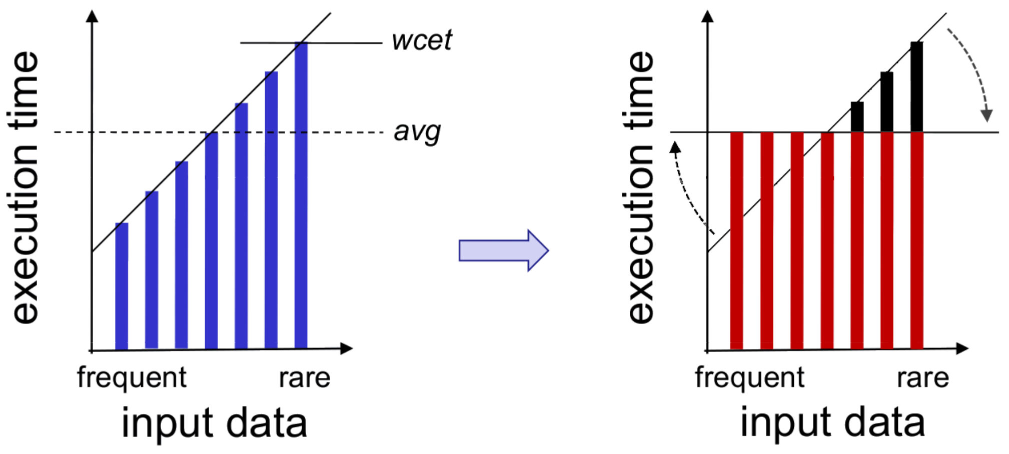


Figure 2.1: Traditional vs. WCET-oriented programming [PK10]

Another important code change is that all loops should have a constant loop bound instead of depending on the input data. Experiments performed in [Pus05] evaluated the difference between code that avoids input dependencies and traditional code. These show that the execution time jitter of the WCET-oriented solution lies in the magnitude of 10%, while the jitter of traditional versions is around 50%. Also the improvements on the WCET path, over the unoptimized version, are between 10% and 20% for the analysed applications.

Finding constant loop bounds is unfortunately not always an easy task. It is not always possible to substitute a constant for a dynamic loop bound. Sometimes it is possible to simply continue performing the same operations although the termination condition in the unbounded loop would have already been met. In some cases this might lead to wrong results. This occurs for example if a data is continuously modified by these instructions or the instructions are performed on an undefined memory area. Also the loop body has to have the same execution time before and after the point when termination condition would have been met.

Other problematic statements that potentially introduce execution time jitter are break/-continue statements. These constructs alter the program flow and often depend on some data properties and should be avoided.

Heuristics that make it less likely for an application to enter the worst case path are also strictly prohibited. Again a good example for such an heuristic is applied on quicksort.

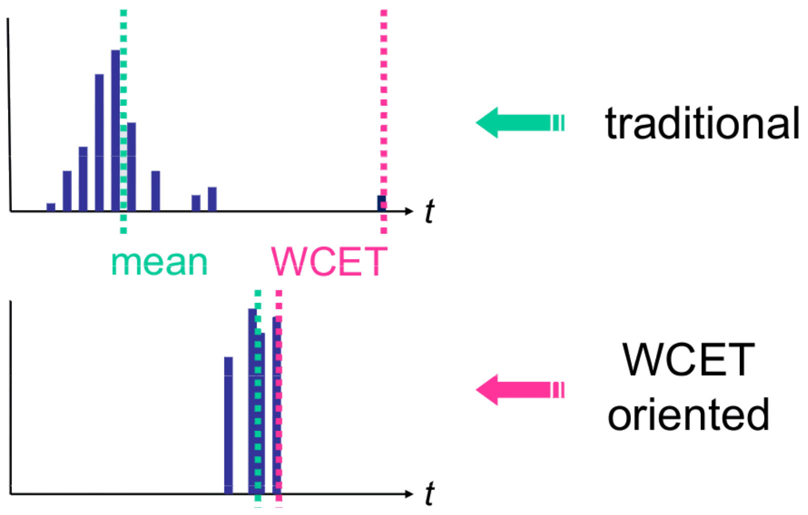


Figure 2.2: Timing jitter in traditional vs. WCET-oriented programming [PK10]

A popular heuristic for pivot element selection is to use the median of three values. This lowers the likeliness of picking a pivot element that leads to the worst case, resulting, on average, in a speed up. This idea however is not suited for real-time systems because the worst case is not improved by this. Analysing the data set at hand and using specific functions that can exploit certain data properties to be faster than a generic function, is also not applicable for WCET-oriented programming.

Avoiding these constructs is good practice in keeping the execution time difference between the best case and the worst case small. In general it also makes the temporal characteristics of the code generally easier to determine. The before-mentioned methods alone cannot, however, be used to decrease the WCET. They more often than not actually increase the WCET. Eliminating data dependencies and algorithms ill-suited for real-time applications must however be a first step to optimize the WCET.

2.4 Programming Optimizations for the WCET

Optimizations for the worst case can be tricky to perform. As a first step the worst case itself has to be determined. Once this has been done and suitable optimizations have been performed it is necessary to find the worst case path again. This is needed to determine whether the optimized path is still the worst case or a different path forms a new worst case. This procedure has to be repeated until a worst case path is found that remains so after the optimizations after which the program can not be improved further. If we find a new worst case path this can either be the case if the previous worst case became shorter that a different path, or because the optimizations are not safe, e.g. the optimization increase other execution paths. Either way the optimizations have to be applied to this path until no new worst case path is found.

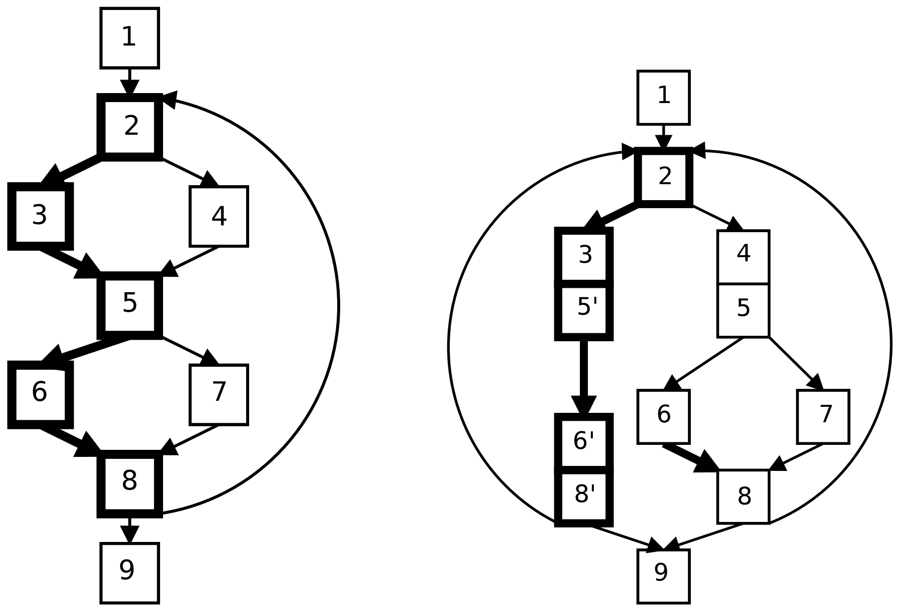
One way to optimize the worst case path is superblock forming [ZKW$^+$05]. The idea is to remove branching code from the longest path by duplicating and splitting code. Removing branching and jumping code decreases the computational overhead. An example transformation is displayed in Figure 2.3. The path in bold marks the worst case path.

Loop unrolling and loop fusion are other safe optimization targeted at reducing the overhead of loops [MM92]. The first method eliminates the loop by sequentially executing all loop iterations without any branches. This obviously increases code size which could have a negative effect in cases where memory constraints are pressing. Loop fusion lowers the overhead by merging loops if their loop headers are identical.

Dead code optimization can be beneficial in reducing the WCET as well [MM92]. This is the case if it improves caching behaviour by not polluting the cache with instructions that are never going to be executed. Removing those potentially results in more useful instructions being kept in the cache. Another speed-up can occur if the code includes branches with invariant conditionals. Removing the test of a condition that always has the same outcome does not alter the semantics of the program but decreases execution time. An optimization that can sometimes be useful is to replace potentially expensive operations by simpler ones like replacing $a = b * 2$ by $a = b + b$. This depends on the specific hardware architecture at hand and can result in a speed-up but it can also have a negative impact on portability.

An example for an optimization that is not always safe, meaning that it can actually end up increasing the WCET by making the execution time of a different path longer, is invariant code motion [MM92]. Invariant code motion moves code, that always has the same result (e.g. an assignment with a constant),from inside loops to a position outside of the loop. This decreases the execution time spend in the loop body by removing the unnecessary assignments that occur if the loop executes more than once. In the case that the loop is never executed, however, execution time is higher than the execution time of the non optimized loop.

All these optimizations are not sufficient to keep the execution time constant. Most of those optimizations can, and should, be implemented in the compiler framework because they are hard to get right on the source code level. Determining the worst case path during development is impossible and all unsafe optimizations therefore not applicable without additional development overhead. Avoiding constructs that harm the WCET will usually make the execution time of all possible paths more equal but may, or may not, improve the WCET. Using optimizations will then lower the execution time on the worst case path resulting in the sketched situation in Figure 2.2 where the jitter and the WCET are lower than in the traditional version.



(a) Program flow (b) Program flow after super block forming

Figure 2.3: Superblock forming [ZKW+05]

One method to equalize the execution time spent on all paths is instruction padding. All paths shorter than the worst case path are extended by inserting NOP (No Operation) instructions until they have the same length. This can however only work on architectures that do not use features like caches, branch predictions etc.. Imagine that one path has a memory access and the other does not. If caches are used the access time to the data will vary and no constant number of NOPs can make up for that difference.

2.5 Single Path Approach

The so-called single path approach described in [PB02] is a method for writing code that has a truly constant execution time. Using code transformations, which can be applied either manually or completely automatic, all input data dependent control decisions are transformed to sequential code with a single execution path, if the code is boundable.

The methods discussed in the previous section did not require any hardware support. The single path approach however requires the CPU to provide predicated instructions. Predicated instructions are instructions whose execution depends on a Boolean value, the predicate, evaluated by the CPU prior to the instruction's execution. If the predicate evaluates to true the instruction is executed, otherwise it is skipped. Fully predicated CPU architectures provide such a capability for all instructions; partially predicated CPU architectures provide at least a predicated move (conditional move) operation. The experiments performed in [Pus05], mentioned in the previous sections, have also evaluated a single path version of the WCET-oriented algorithms. The execution times were between 10% lower and 10% higher than the WCET optimized version, showing that the single path approach does not negatively impact the execution times by a prohibitive large amount.

In the following paragraphs we are going to describe the transformations used by this approach. The formalization for the translations is taken from [Pro08]. The execution of every operation depends on some precondition. Only input dependent statements have to be transformed because non input dependent statements do not contribute multiple paths. To formalize the translations we define a function $ID(statement)$ that returns true if the statement is input dependent and false if it is not. We use δ as a counter to create unique predicates subsequently and σ as the inherited precondition from previously transformed code constructs.

The translation is defined as a recursive function $SP[[p]]\sigma\delta$ where p is the code that is being transformed. Figure 2.4 demonstrates the translation of a single statement. If the predicate is always true it is executed, if it is always false it is removed. If the value of the predicate is not known before hand it will be evaluated during execution.

$$\text{SP[[S]]}\sigma\delta \Rightarrow \begin{cases} S & \text{if } \sigma = t \\ & \text{if } \sigma = f \\ [\sigma]\text{S} & \text{otherwise} \end{cases}$$

Figure 2.4: Transforming a sequence of statements

Th *if-translation* displayed in Figure 2.5 uses the condition as an additional predicate called *guard*. It transforms the statements of both bodies using the *guard* and σ as the predicate for the translation of the statements. Depending on the precondition and the *guard* one of the conditional bodies is executed.

$$\text{SP[[if } \mathit{cond} \text{ then } S_1 \text{ else } S_2 \text{ endif]]}\sigma\delta \Rightarrow$$

$$\begin{cases} \begin{array}{l} guard_\delta = cond; \\ \text{SP}[[S_1]](\sigma \wedge guard_\delta)(\delta+1); \\ \text{SP}[[S_2]](\sigma \wedge \neg guard_\delta)(\delta+1); \end{array} & \text{if } ID(cond) = t \\ \\ \begin{array}{l} \text{if } cond \text{ then} \\ \quad \text{SP}[[S_1]]\sigma\delta \\ \text{else} \\ \quad \text{SP}[[S_2]]\sigma\delta \\ \text{endif} \end{array} & \text{if } ID(cond) = f \end{cases}$$

Figure 2.5: Transforming an if statement

The *loop-transformation* displayed in Figure 2.6 uses the termination condition to set a guard end_δ depending on whether the loop has to execute the body or not. It inserts a conditional statement at the beginning of the loop (which now executes exactly N times instead of at most N times). This conditional is then subsequently transformed using the *if-translation*. The loop body is then executed depending on the value of the end_δ guard set by the transformed conditional. If the loop is not input dependant only its body is transformed.

$$\text{SP}[[\text{while cond max N times do S endwhile}]]\sigma\delta \Rightarrow$$

$$\begin{cases} \begin{array}{l} end_\delta = t; \\ \text{for } count_\delta = 1 \text{ to N do} \\ \quad \text{SP}[[\text{ if } \neg cond \text{ then } end_\delta = t \text{ endif}]]\sigma(\delta+1); \\ \quad \text{SP}[[\text{S}]](\sigma \wedge end_\delta)(\delta+1); \\ \text{endfor} \end{array} & \text{if } ID(cond) = t \\ \\ \begin{array}{l} \text{while cond max N times do} \\ \quad \text{SP}[[\text{S}]]\sigma\delta \\ \text{endwhile} \end{array} & \text{if } ID(cond) = f \end{cases}$$

Figure 2.6: Transforming a loop

$$\text{SP}[[\text{funcname(parameters)}]]\sigma\delta \Rightarrow \begin{cases} funcname(parameters) & \text{if } \sigma = t \\ funcname_{sp}(parameters, \sigma) & \text{otherwise} \end{cases}$$

Figure 2.7: Transforming a function call

The transformation of functions and their calls is demonstrated in Figure 2.8 and Figure 2.7. Function calls within input dependent code segments have to use the condition from before the function call as an additional function parameter. The function body is then transformed using this as an additional guard.

$$\text{SP}[[\text{func f(parameters) S end}]]\sigma\delta \Rightarrow \begin{array}{l} \text{func } f_{sp}(parameters, precond) \\ \quad \text{SP}[[\text{ S }]](precond)(0) \\ \text{end} \end{array}$$

Figure 2.8: Transforming a function

CHAPTER 3

Related Work

Introduction

In the course of this chapter we will explain some required terms and explore the basic ideas that lead to this project. The first part we present an overview of concepts important for real-time systems. It discusses different kinds of control transfers and task structures and how they are related. This is followed by an investigation of scheduling decisions and inter-process communication. These each include a short section explains the temporal characteristics of these methods. The chapter is concluded by a discussion of WCET analysis and work related to this project.

3.1 Real-time System Classifications

Before talking about RTOS concepts it is important to discuss how control can be exercised and the structure tasks can have. These are important system properties and influence the operating system structure greatly.

3.1.1 Control Signals

Control signals are entities that instruct an application to perform certain actions like sending messages, activating tasks etc. One possible way to generate control signals is based on events in the environment, or the computer system itself. A typical example for an event is an interrupt. These events do not occur at predetermined points in time but occur sporadically. These event-triggered systems experience high flexibility and usually a fast response time because they can react to inputs fast. They are however harmful to the temporal analysability of real-time systems as a result of the indeterminism of the frequency and instant of time when events occur.

Another category consists of time-triggered systems. These kind of systems base all control decisions on the progression of time. The benefit is that due to the complete determinism temporal analysability is easier. The drawbacks are a lower flexibility and potentially higher response times. This is a result of the problem of not being able to detect external events exactly when they occur, but only at predefined time at which the system can check the condition [Kop97]. Naturally a combination of both types is possible to increase flexibility and suitability to certain scenarios.

3.1.2 Tasks

“A task is the execution of a sequential program. It starts with reading of the input data and of the internal state of the task, and terminates with the production of the results and updating the internal state” [Kop97].

A task that does not contain any synchronization points in its body is called a simple task (S-task). Started S-tasks can run to completion without having to explicitly wait for computational results of other tasks. The response time of a S-task hence is only a function of its own computations. The worst case response time of such a task is equal to its WCET, but can be increased by indirect interactions introduced by task preemptions. These prolong the execution time of the task but do not change the time required for individual operations executed in the body [Kop97]. Figure 3.1 shows typical states a S-task in a time-triggered system can assume. A task is running when it currently has control of the CPU, ready when it was interrupted by a different task and inactive if it has not started execution yet.

A task that contains blocking synchronization statements like semaphores within its body is called a complex task (C-task). Such a C-task requires actions on resources shared with other tasks. The risk of conflicting accesses makes it necessary to guard the resource against undefined *read/write* or *write/write* access patterns. The amount of time such statements have to prevent the task from accessing the resource depends directly on the behaviour of other tasks in the system. The response time of a C-task therefore includes not only the WCET of the task, but also time time for all computations of other tasks that are performed while it is in the blocked state. This constitutes a global, and not just local problem, making temporal analysis much harder if not even impossible [Kop97].

Figure 3.2 shows typical states a C-task in an event-triggered system can assume. A task is running when it currently has control of the CPU, ready when it was interrupted by a different task and inactive if it has not started execution yet. New is the suspended state. A task is “blocked” when it is waiting for access to a shared resource that is currently in use by someone else or not yet produced. Once this is produced or the task blocking it left its critical section (the code section where the task accesses the resource) all “blocked” tasks transition to the ready state.

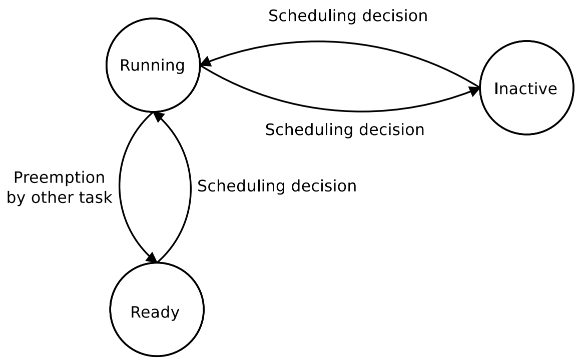


Figure 3.1: State transitions in TT systems with preemptive scheduling and S-tasks

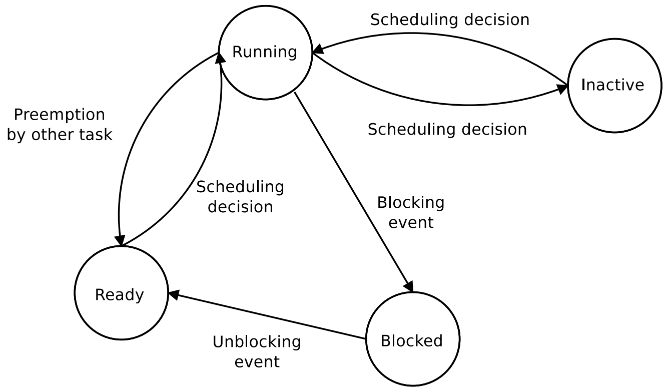


Figure 3.2: State transitions in ET systems with preemptive scheduling and C-tasks

3.2 Real-time Operating Systems

Real-time operating systems (RTOS) are still an active research area and many different implementations, tailored for specific needs, exist. Some are extensions to non real-time kernels like Linux or MACH while others are custom made for specific applications or with a specific design goal in mind [SR04].

Real-time extensions for popular operating systems like Linux have the benefit of leveraging the flexibility provided by the vast ecosystem they supply. Reusing methods and coding styles inherited from the non-real-time world unfortunately limits the predictability of the operating system. This is fine as long as only soft real-time requirements are demanded but can be detrimental for hard real-time systems. Workarounds improving this have been implemented to various degrees, but these come at the expense of higher complexity, or have to sacrifice flexibility again.

A different approach to RTOS is developing small kernels specialized to the applications needs. These kernels are typically intended for small embedded systems where resource constrains forbid the use of the more complex RTOS based on non real-time operating systems. They often follow a micro kernel approach providing only the absolutely necessary services needed by the system. Some of the typical characteristics RTOS usually are usually optimized to provide given in [SR04] are:

- Fast context switch
- small code size
- fast response to interrupts
- minimize intervals during which interrupts are disable

Features they most often provide are for example [SR04]:

- fixed or variable partitions for memory management
- provide primitives for IPC and synchronization
- support multi tasking and priority-based preemptive scheduling
- bounded execution time for most primitive

As a minimum a RTOS has to provide mechanisms for scheduling, inter-process communication and/or synchronization as well as some input/output mechanism. Even today RTOS features are often simply designed to be “fast” although this is not sufficient for real-time systems [SR04]. Processing speed does not guarantee that deadlines are met. However for some simple applications it can potentially still be possible to demonstrate that all timing requirements are met. The more complex a system gets the less suited such an approach becomes.

Much research has gone into replacing mechanisms that are merely “fast” by boundable and predictable mechanisms. The following sections will describe some approaches that are commonly used for scheduling and communication in RTOS. These include approaches from both the “fast” as well as the “predictable” category.

3.2.1 Scheduling

“The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput and processor efficiency.” [Sta01]

For real-time systems the primary objective is that every process (task) finishes its execution before its deadline. Many scheduling problems are known to be NP-complete, for example if multiple processors exist, or in certain configurations with shared resources and precedence relationships. Many heuristics and approximations have been developed that deal with these constraints in a computationally traceable way. Still special care has to be applied concerning scheduling algorithms owing to their suitability depending heavily on the specific scenario [RS94] [Kop97].

Scheduling decisions can either be taken online, during the system execution, or be determined offline, before system execution. If the scheduler can only start a new task after the complete execution of a currently executing task it is called non preemptive. The policy that allows invoking the scheduler at any arbitrary point in time is called preemptive [Sta01]. The decision which task to execute next is often based on priorities assigned to the tasks either statically or dynamically, depending on the system environment.

A rough classification of the most important real-time scheduling approaches has been given in [RS94]:

- Static table-driven: A static schedulability analysis is carried out prior to system run-time and a resulting schedule (stored in a table) is read during run-time.

- Static priority-driven preemptive: Again a static schedulability analysis is performed during system design, but during run-time tasks are selected based on the highest priority instead of using an explicit table.
- Dynamic planning-based: Feasibility analysis is performed during run-time when new tasks become ready for execution. Ready tasks are only selected if it is feasible that all tasks meet their deadlines.
- Dynamic best effort: No feasibility analysis is performed at all. It is not guaranteed that tasks meet their deadlines.

Priority-driven preemptive scheduling is one of the most common approaches used for real-time and non real-time systems. The priority is typically based on the deadlines the tasks need to fulfil. This can be done either statically, which means that the priority of a task never changes, or dynamically where the system state and other tasks are taken into account to determine the tasks' priorities.

The classic algorithm for determining schedulability using static priorities, is the rate monotonic algorithm. It allows the scheduling of periodic tasks, which are strictly periodic and experience neither precedence constraints nor resource sharing. Priorities are based on the task periods and the tasks with the shortest period get the highest priority.

Assigning the priorities statically has the benefit that it has a low run-time overhead and once the schedule is found to be acceptable it does not have to be re analysed. The FreeRTOS operating system is an example for an RTOS that uses this mechanism [Bar09].

Dynamically assigning the priorities on the other hand can be used to schedule for sporadic tasks, as well at the cost of a higher overhead.

Dynamic planning based algorithms perform a feasibility check every time a new task is ready to execute. Tasks are guaranteed to meet their timing constraints only if the assumptions the algorithm relies on hold. These assumptions could be about the tasks WCET, resource needs, etc.. If it is found to be feasible under the current system state the schedule is modified accordingly.

This approach provides the highest flexibility especially in mostly event-triggered scenarios, but requires the highest computational overhead [RS94]. A similar approach, typically with lower overhead and complexity is the dynamic best effort scheduling. It does no feasibility checking at all but its scheduling decisions are based on simple criteria and it simply hopes to satisfy all deadlines. The criteria the priority is calculated from are similar to static priority and include for example FIFO, earliest deadline first or even random. Confidence in the temporal correctness has to be gained by testing and simulation, typically non exhaustive approaches, which clearly constitutes a drawback.

If the requirement that all deadlines must be met is the primary criteria for the scheduler, as is usually the case in real-time systems, the static table approach provides a priori knowledge about the feasibility of the schedule. During system design heuristics can be used to construct a schedule that allows every task to meet its deadline. This information is stored in a scheduling table that is evaluated by the operating system during run-time. This table contains all the times at which tasks are activated. During run-time the scheduler cyclically reads this table and starts the task at the next table entry. If a schedule with the length of the least common multiple of all tasks periods can be found it can be concluded that it is feasible. The requirement for this to work is that all tasks have to be periodic. This means that sporadic tasks have to be transformed to periodic ones. This is possible if the worst case inter arrival periods of these tasks are known. Using this knowledge a periodic task that runs exactly at these points in time can be created to service these events [Kop97]. The benefit of this approach is the complete predictability of the system but at the expense of flexibility and utilization. As a result of the schedule's feasibility being known before system run-time no such checks have to be performed during the execution of the system. This guarantees that the run-time overhead of the scheduler is low.

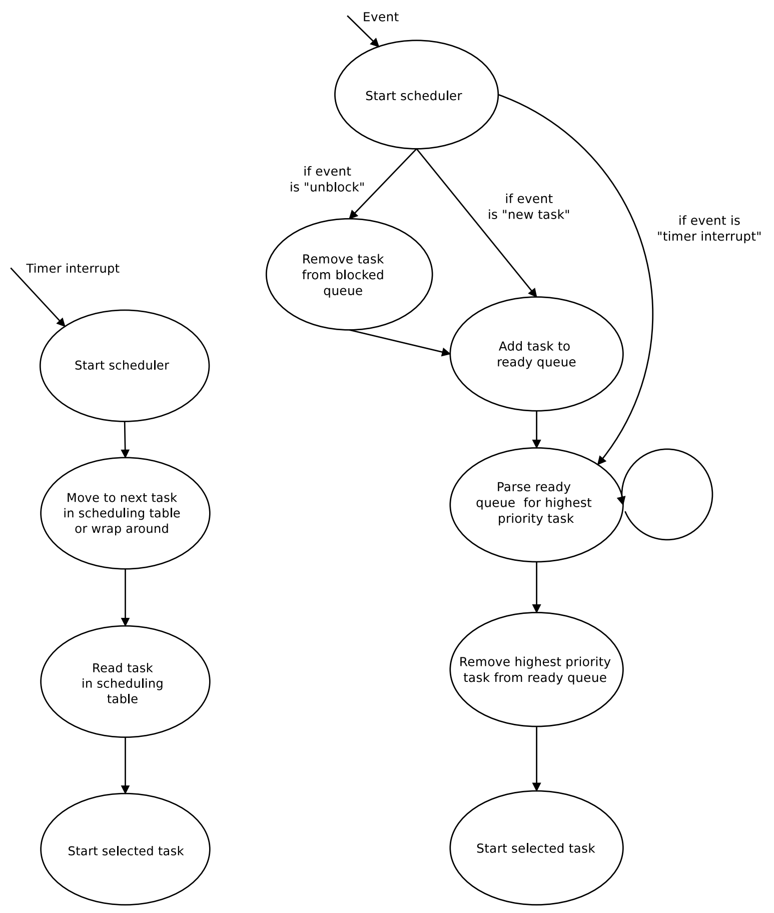
Temporal Properties

The usefulness of the different approaches depends on the characteristic of the system. The dynamic best effort approach is reliably applicable only to soft real-time system. Its low overhead during run-time it make it well suited if objectives like throughput or processor utilization are important. Missed deadlines are however only recognized after they are already missed [RS94]. Many, even real-time, applications use best effort scheduling due to its low computational overhead. If resources are limited and the system complexity is low enough to conclude that all deadlines are met nevertheless [SR04].

As systems grow more complex and the timing requirements become more stringent this approach becomes less suited as a result of the lack of predictability. This makes it is hard, or even impossible, to conclusively demonstrate the correctness of the system. Dynamic planning approaches can guarantee the timing requirements but their higher complexity in both the time and functional domains make them harder to analyse compared to the other approaches.

Static priority-driven and static table provide a-priori knowledge about the feasibility of the schedule. Already before the system executes it is known if all deadlines can be met. The static table based scheduler does not need to find the task with the highest priority but simply takes the task at the next position in the table. This leads to a lower complexity and hence potentially a simpler analysis.

Figure 3.3 shows typical decisions taken in time-triggered and static priority based scheduling. Although a scheduling approach using static priorities is already well suited for real-time systems and has a very predictable structure it can still be seen that the time triggered approach has an even simpler structure, that makes it more suited for keeping the execution time constant. Dynamic scheduling approaches would be even less suited for the overall timing behaviour. The dynamic nature of the priority generation also makes the scheduler itself more receptive for jitter in its execution times.


Timer interrupt
Start scheduler
Move to next task in scheduling table or wrap around
Read task in scheduling table
Start selected task
Event
Start scheduler
if event is "unblock"
if event is "new task"
if event is "timer interrupt"
Remove task from blocked queue
Add task to ready queue
Parse ready queue for highest priority task
Remove highest priority task from ready queue
Start selected task

(a) Time-triggered scheduling (b) Priority driven scheduling with C-tasks

Figure 3.3: Scheduling decision comparison

3.2.2 IPC and Synchronization

Inter-process communication (IPC) is, as the name implies concerned with the data exchange of (concurrently) running tasks. If access to a common resource is required, synchronization between the accessing tasks has to be ensured to prevent conflicting access patterns like interrupting a read operating by a write.

Messages can either contain state or event information. Event information is used to update a state piecewise and therefore must be read exactly once and strictly in order of arrival. As an example for such an event information consider a node with a temperature sensor that always sends only the difference to the previous measurements to a different node. The changing of the temperature constitutes the event. The receiving node then uses this value to update its own temperature variable. If such a message is read multiple times the receiving node will update its variable too often leading to a wrong value. Reading these messages out of order will result in the receiving node having a wrong value until all outstanding messages are read.

State information contains information about a state, in our temperature reading example this would be the current temperature. If the sending node always sends the current value read from the sensor and not just the temperature difference to the previous measurement, reading such a message multiple times causes no harm. Also old versions of the message can be dropped as soon as a newer one is available.

Data is exchanged either using message semantics, which requires some identification mechanisms to identify sender/receiver or the message itself, or using common data areas which constitutes a more indirect message exchange mechanism [Kop97]. For the latter case special care has to be taken that tasks do not concurrently access this shared resource and data corruption can occur. For these explicit approaches like implementing the synchronization with the application code, for example with the Dekker algorithm, or operating system provided semaphores can be used [Dij02].

Software approaches to the problem of shared resources are usually very error prone, incur a high computational overhead and are generally considered not suited for real applications. Therefore operating systems commonly provide semaphores to guard shared resources against conflicting concurrent accesses.

Semaphores are simple data structures. They typically consist of a counter, indicating how many tasks are allowed to access the shared resource guarded by the semaphore, and a queue specifying the order in which blocked tasks can access the resource once it is available again. Two functions, *wait* and *signal* are used to access the value. The *wait* function is called before a task enters the critical section, the section of its code during which it accesses the resource. This function forces the task to wait until the value of the semaphore is ≥ 0 (the resource is ready to be accessed) and places the task in the associated queue if it is ≤ 0 (the resource is not ready to be accessed). Usually this *wait* function invokes the scheduler which starts a different task that is currently ready for execution.

Tasks leaving their critical section call the *signal* function which increases the semaphore value again. This would allow the next tasks in the semaphores queue to enter its critical section once it is executed again. This *signal* moves the first task currently waiting for the resource in the queue from the suspended to the ready state as sketched in Figure 3.2.

For processor utility reasons a context switch is typically executed whenever a task is blocked to not waste processor time until the other tasks release the semaphore [Sta01].

A scenario where a task with a low priority blocks a task with a higher one, effectively inheriting the higher tasks priority, is called priority inversion. To handle shared resources with semaphores the priority ceiling protocol is used on top of the rate monotonic algorithm. A task can access a shared resource if its priority is higher than the priority of all tasks that *may* access currently locked resources. If it can access the resource it executes its critical section either using its own priority, or inherits the highest priority of all task that it blocks It can be shown that this way no low priority task can block a task with a higher priority [Kop97] [Sta01] [RS94].

Another useful mechanism is message passing. It usually consists of 2 functions, send and receive that require the destination, respectively the source, to specify which message to transmit and read. Obviously a task can only read a message that has actually been sent, therefore synchronization is needed between the sender and receiver. Possible options are [Sta01]:

- Blocking send and blocking receive: Both tasks are blocked until the message is delivered.
- Nonblocking send and blocking receive: The receiver waits until the message is completely transmitted but the sender can carry on without waiting for the message arrival.
- Nonblocking send and nonblocking receive: Neither wait for the message delivery

If the sender-receiver relationship is not one-to-one and/or multiple messages can be sent before they are consumed, messages have to be stored in a queue to prevent message loss . Assuming every time that, to reach an up-to-date system view, an amount of messages equal to a maximum queue depth has to be read is detrimental for the WCET analysis. If there is no maximum queue depth the WCET is even not determinable when event messages are used.

The problem with all of these approaches is that it either places a burden on the schedulability checks, semaphores experience this problem, or message queues are used which potentially lead to a high WCET overestimation. A solution to the reader/writer problem using a shared memory area that provided non blocking send and receive operations is

```
Initialization:
    $CCF_i := 0;$
Write message i:
    $CCF_{old} := CCF_i$
    $CCF_i := CCF_{old} + 1$
    <write $buf_i$ >
    $CCF_i := CCF_{old} + 2$
Read message i:
    do {
        $CCF_{begin} := CCF_i$
        <read $buf_i$ >
        $CCF_{end} := CCF_i$
     } while($CCF_{end} \neq CCF_{begin} \vee CCF_{begin} = odd$);
```

Algorithm 3.1: The NBW Algorithms [KR93]

the Non Blocking Write Protocol [KR93]. The NBW Protocol assumes a 1-to-n relation between sender and receiver using a shared memory. The algorithm is shown in Algorithm 3.1. Before and after a writer accesses a buffer it increments a global flag variable. A reader tests this variable before and after it reads the buffer. If the value is the same both times it knows that no write occurred during the read and the content is uncorrupted. If however the value changed in between it tries to read again. Kopetz and Reisinger showed that it is possible to bound the number of times a reader has to reread due to writer interference.

Another approach is the Double Buffer Algorithm presented in [HS02]. The algorithm is shown in Algorithm 3.2. It uses a 2-dimensional array with $\#tasks + 1$ rows. Every row has a variable associated with it indicating the number of readers currently accessing the buffer ($ReaderCnt$) and a flag indicating which buffer contains the most recent message (Cl). Also a pointer $latest$ indicates the most recent buffer. The writer parses the row for an $ReaderCnt$ entry that has no reader currently accessing its buffer. It then writes to the older of the 2 buffers associated with this row as indicated by the Cl flag. It subsequently updates the flag indicating that this buffer is now more recent and sets the pointer $latest$ to the new buffer.

```
Initialization
    int NReaders;
    int Nrows = NReaders +1;
    int Latest;
    message Buff[NRows][2];
    int ReaderCnt[NRows];
    boolean CI[NRows];
$Reader_i$()
    ridx := Latest;
    ReaderCnt[ridx]++;
    cl = Cl[ridx];
    $read$ Buff[ridx][cl];
    ReaderCnt[ridx]–;
$Writer_i$()
    for(i=Latest;;i++) {
        if(ReaderCnt(i mod NRows] == 0 break;
    cl = not Cl[i];
    $write$ Buff[i][cl];
    Cl[i] = cl;
    Latest = i;
    }
```

Algorithm 3.2: The Double Buffering Algorithm [HS02]

Temporal Properties

Especially semaphores, but also message queues, are hard to analyse in the temporal domain. Analysing the access pattern to a shared resource with semaphores is a global problem, potentially encompassing all tasks and all resources in the system. Determining the temporal behaviour of a task in isolation is therefore impossible. Scheduling analysis with shared resources and precedence constraints is very hard and algorithms for scheduling table generation have to be evaluated carefully. Message queues can in certain cases be easier to analyse because the execution time of a sending task does not depend on the receiver. Reading all enqueued messages, however, depends on the relative speed of sender/receiver.

Using non blocking write protocols makes it easier to reason about their execution times and schedulability. Especially in time-triggered systems tasks do not have to spend an unknown time waiting for other tasks to finish their critical section and can be analysed in isolation. Huang et al. [HPS02] compared some approaches, including the NBW and Double Buffer Algorithms, and showed that these approaches provide a far better worst case behaviour than locking based ones. Furthermore they found out that the execution time jitter is much lower. The retry mechanism of the NBW protocol makes it impossible to reach a constant execution time. Double Buffering write access execution depends on which buffers are in use but this could probably be made constant by using a bounded loop. As a result both mechanisms are not suited for keeping the execution time constant but Double Buffering can, with additional memory overhead, be adapted to this end.

3.3 WCET Analysis

There are two methods for analysing the temporal characteristics of an application. [PK10] Generating exhaustive test data and measuring execution traces to determine and quantify the worst case path on the one hand. A static analysis that analyses the source code to determine the worst case path and give an upper bound for the its execution time on the other.

Generating all input data is infeasible for all but the simplest applications, therefore only an interesting subset of the possible input data is generated. This data generation is often guided by heuristics like evolutionary algorithms. Given the right tool support this can be done quite efficiently, but incorporates the risk of not actually finding the real worst case path which leads to underestimating the WCET, at the risk of missing deadline violations. For soft real-time systems this method is in general suited well enough but clearly is not applicable for hard real-time systems.

Static analysis on the other hand can not be completely automated, but requires extensive support from the developer by specifying timing characteristics of the hardware platform and providing annotations helping the tool extract information from the source code. This is a time consuming and error prone task, but if done right provides a safe upper bound on the WCET of the system. Although extensive research has been carried out over the last two decades the size of the problems that can be analysed remains fairly small but is already good enough for many real-time applications [WEE$^+$08] [PB00].

3.4 State of the Art - RTOS and Analysability

FreeRTOS [Bar09] is an example for a RTOS targeting embedded systems with a small resource foothold. Its main focus is not on temporal predictability but on functional correctness and small computational overhead. FreeRTOS uses a static priority scheduler, which is periodically invoked by a timer interrupt. It provides semaphores and queues for synchronization and IPC. Given the unpredictable nature of these features, conclusively arguing about the temporal safety of applications developed using this RTOS is therefore not a trivial task.

RTEMS [Cor03] is another RTOS that caters to applications with stringent real-time requirements. It offers similar mechanisms as FreeRTOS and the analysability by a static WCET tool was determined in [CP01]. Apart from hard to analyse features like semaphores, being a schedulability issue which is not a concern in this paper, some parts of the operating system were found to depend on run-time properties. The scheduler for example can experience different behaviour depending on what kind of interrupts can occur in the system and the occurrence of the timer interrupt.

Examples for operating systems focusing specifically on temporal predictability are the MARS and the Spring operating systems [Rei93] [SR04]. The MARS operating system employs a strictly time-triggered approach which performs scheduling and IPC only at predefined times. A special task is invoked to copy IPC data from privately owned memory to a global buffer. This implements the state-message semantics proposed by Kopetz [Kop97] and allows precise control over the time when data is produced. As discussed before this reduces flexibility but has the benefit of complete a-priori knowledge about the temporal characteristics of the system. The operating system was not optimized to have a constant execution time nor was it specially WCET-optimized beyond avoiding unpredictable constructs.

Procter and Shackleford [PS01] demonstrated that execution time jitter is not merely a problem for temporal analysability but that some applications are impacted negatively by the jitter as well. They experimented with a motor control application using a RTOS running on an x86 CPU. The jitter negatively affected the torque load which could cause a loss of position steps. This further demonstrates the need of research on WCET oriented programming and more predictable hardware architectures.

In recent years many RTOS were analysed using static WCET tools [LGZ$^+$09] [SEGL04]. The focus on static tools was mostly due to the safe upper bounds they provide compared to measurement based methods. Also in some cases another goal was to determine the effort in applying them to operating system code. Many constructs were found to be hard or impossible to analyse and manual intervention was needed to gain results on most analysed functions. Depending on the method used by the tools unstructured control flow (e.g. goto statements and loops with multiple exits) presented a problem. Recursions pose another problem for some methods as well. Implicitly this means that WCET-oriented programming techniques had to be applied to the code first to get results at all. Especially troublesome seemed to be a close connection of system parameters and run-time properties, which either require manual annotations or are unboundable in general. Also context switch times can depend on the time of interrupt arrivals which are unpredictable by nature. The impact of blocking statements like semaphores were outside of the scope of these papers as long as they were not used by the operating system functions. Being a schedulability issue they do not contribute to the WCET of a task, only impacting the overall timing behaviour of the system.

Khyo et al. [KPD08] demonstrated that it is possible to design an operating system that experiences a constant execution time. They closely followed the concepts for scheduling and IPC demonstrated in the MARS operating system and combined it with the single path approach using a custom made hardware architecture optimized for real-time systems. [DHPS03] The results determined by measurements show that the execution time of the demo application only depends on parameters known at compile time. This is in line with the goal of having a-priori knowledge of the execution time. They also performed a small experiment removing the single path transformations from parts of the code to determine the jitter. They however did not explore if the resulting jitter could be removed with code changes following the WCET-oriented programming approaches.

This work differs from the operating systems developed by Reisinger [Rei93] in that [Rei93] did not focus on the WCET analysability of the operating system and tasks. Reisingers prime objective was to evaluate the suitability of the time-triggered approach for operating system design. Khyo's [KPD08] operating system on the other hand required the single path approach and used specially optimized hardware while this work is based on WCET-oriented programming paradigm, that does not require special CPU support.

CHAPTER 4

Infrastructure Description

Introduction

For reasons of reproducibility this chapter gives a short overview of the hardware and software used in this project.

4.1 Development Board

The hardware used for the development of the operating system is a Nios Stratix II Development Board from Altera. The board provides a Stratix II FPGA which is big enough for many different configurations of the LEON3 CPU design and provides the following features that are used by this thesis for communication and debugging:

- RS232 serial port
- Push buttons
- LEDs
- 2 7-segment displays
- Prototype headers for general purpose IO

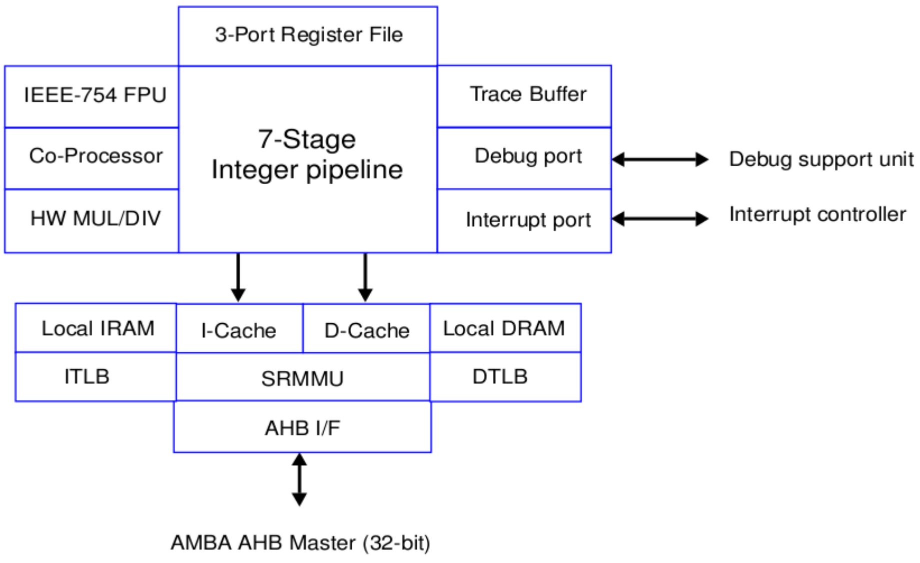


Figure 4.1: LEON3 high level block diagram [Gai10]

Two boards are used to demonstrate IO operations. The boards are connected via the RS232 port for data transfer and a GPIO pin that is used to start the execution on the slave board.

4.1.1 CPU Core

The CPU is a LEON3 soft-core developed by Aeroflex Gaisler. This CPU implements the SPARC V8 instruction set with the V8e extension. It is a 32-bit RISC instruction set developed to be scalable from small embedded applications up to servers [Inc92]. It is highly configurable with options ranging from no caches at all to different cache sizes and replacement policies or scratchpad memory to an MMU and hardware multiplier and a floating point unit. For expansion it uses an AMBA2 bus and Gaisler provides many modules with its grlib collection of IP cores [Gai10]. Most designs are available under the GNU GPL license.

The CPU implements a 7 stage in order pipeline with the instructing timings listed in Table 4.1.

Instruction	Cycles
Double load	2
Single store	2
Double store	3
Taken Trap	5
Atomic load/store	3
SMUL/UMUL	4
SDIV/UDIV	35
all other instructions	1

Table 4.1: Instruction timing

The design used for the measurements has the following characteristics:

- No instruction and data caches
- Local instruction and data scratchpad memory
- No MMU
- No FPU
- UART module
- Debug Unit
- 4 Timers
- 32 bit GPIO module

Other designs that used caches and/or the DDR2 RAM available on the development board were used and tested during development. Due to the indeterministic nature of these hardware features these were not used for measurements. For real-world applications the scratchpad memory alone will most likely not be large enough, therefore using the DDR2 memory is a viable option if memory requirements are too high. Synthesising a working design with the DDR2 controller required the use of Altera Quartus in version 7.1 and the option “Allow Synchronous Control Signals“ had to be disabled. Neither higher versions nor only using this option did suffice to get it to work. The reason for this could not be determined.

4.2 Software

The build infrastructure used is the Bare-C Cross-Compiler System for LEON provided by Aeroflex Gaisler in version 1.0.36b. As compiler it uses a SPARC v8 port of the Gnu C Compiler version 4.4.2 and provides a libc implementation called newlib.
For IRQ handling and for debugging single node applications also the stdio functions of the newlib C library are used. The context switch code has been adapted from the SPARC v8 port of the FreeRTOS operating system and also uses macros provided by the newlib library (version 1.13.0-1.0.31) [Gai10] [Bar09].

For developing and debugging the Eclipse CDT plugin in combination with the grmon-eval tool also provided by Gaisler was used.

CHAPTER 5

Overview of the Operating System

Introduction

In this chapter we are going to describe the overall structure of the operating system and give a rough overview of the concepts implemented in this project. It is explained why the chosen concepts are suited for temporal predictability and for providing constant execution times. We introduce the different execution phases and explain the need for this distinction. We give a high level description of the task structure, scheduler, IPC operations and the I/O mechanisms to explain why these concepts were choses. The details of the remaining modules and the specific implementation is given in Chapter 6.

5.1 Structure of the Operating System

The operating system provides mechanism for scheduling, task management, two slightly different IPC mechanisms, a time-triggered I/O module and simple memory management. As mentioned in Chapter 2 these are the required mechanism a RTOS has to provide. The high level view of the operating system's components is displayed in Figure 5.1. Using a micro-kernel approach many mechanisms are outsourced to tasks. Having only the features that are absolutely essential as a direct part of the operating system code ensures that the complexity of the operating system stays low.

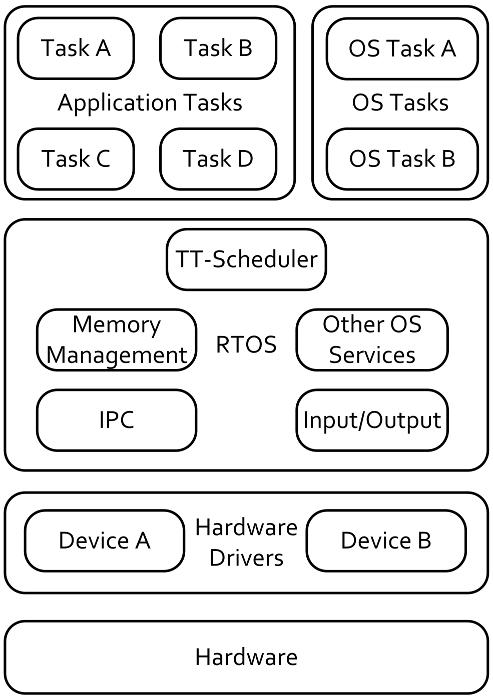


Figure 5.1: Structure of the operating system

The goal of this project is to develop an operating system with good temporal predictability on the system level as well as each function having constant execution times. The algorithms implemented hence cannot use blocking statements and no decisions are allowed to be based on run-time properties. Subsequently the implementation of these algorithms must be carried out in a way that ensures that the execution times are constant or at least have only a small jitter.

The scheduler is based on a time-triggered static table approach using S-tasks as described in Chapter 2. No blocking statements are required, guaranteeing temporal isolation of the tasks by the scheduler. As a result the response time of the tasks is equal to its WCET plus the overhead introduced by preemptions.

The system is executed cyclically. This means that its execution never stops and tasks are started and interrupted only based on the progression of time. All tasks have to be created at the beginning of the system's execution and every task has to be cyclic, meaning that they can never return or exit. There are no sporadic tasks in the system. This is needed to ensure the strictly time-triggered nature of the operating system. As a result of the a-priori knowledge of all the instants of time tasks are starts, good temporal analysability is guaranteed. Completely stepping through the scheduling table once constitutes one *execution round* of the cyclic system execution.

Changes in the task state do not require scheduler activation, as would be the case if a task executes a wait operating on a semaphore, leading to predictable behaviour at the system level. Also the low number of decisions guarantees a low execution jitter in the scheduler functions.

The IPC mechanisms are based on a shared memory approach which offers a data exchange mechanism with low overhead, based on state messages.

When using the state message approach the number of messages to read depends only on the number of tasks that communicate with the receiver. When using event messages instead messages would have to be enqueued and the execution time would additionally depending on how often a sending task sends a message to the receiver. The receiving task would be required to read all messages that were sent by every sender since the task's last execution, instead of only the last message from each sender. A shared memory approach with state messages hence offers better temporal predictability than using message queues and event messages.

The operating system follows a micro-kernel approach by executing as many operations as possible in tasks that do not differ from tasks created by an application programmer. The following tasks are provided by the operating system and can be used depending on the specific configuration.

- IPC send operations can either be executed directly by the sending task or are scheduled at a later time using the *ipc_send_msg* task structure.
- A simple clock synchronization task, *wait_sync_msg* is provided that exchanges clock information and brings the clocks closer together following a client/server approach.
- Send and receive I/O operations accessing the communication medium are contained in corresponding tasks called *ttio_send_msg* and *ttio_receive_msg*.

Chapter 6.8 describes the details on how to configure the system.

Furthermore the operating system prints error messages, either to a console, or displays an error code on the LEDs and 7-segment displays. The list of error codes and their meanings is included in Appendix A.

5.2 Task Structure

Each task must consist of a pair of functions, an *initialization function* and a *task function* that serves as the main function of the task and implements the desired functionality. The *task function* has to encapsulate all operations within an infinite loop whose first operation has to be a context switch to the *idle* task. This is needed to prevent the execution of code meant for the next execution round by entering the next loop iteration prematurely. The tasks are based on the S-task model, which means that tasks are never blocked and that all input is available at the time the task execution starts. The indeterminism introduced by the required blocking mechanisms prohibits the application of C-task. Each task has its own stack which is used to store its local variables and the context whenever it is preempted by the scheduler. A rough overview of the structure of a task is given in Figure 5.2.

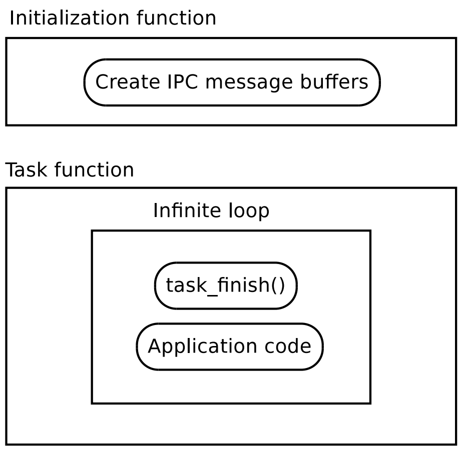


Figure 5.2: Structure of tasks

5.3 Execution Phases

The execution of the system is split into two phases, the *initialization phase* and the *real-time phase*. The *initialization phase* is executed first to perform operating system and other initializations and the *real-time phase* is executed subsequently and is never exited. During the *initialization phase* no function has to have constant execution times, nor are any operations required to meet deadlines. As a result of not having real-time requirements these functions also do not have to be analysed for their temporal properties.

After all initializations have been carried out the *real-time phase* is entered during which the tasks are executed cyclically and timing has to be strictly bounded. Figure 5.3 shows the division into these phases and sketches what actions they perform.

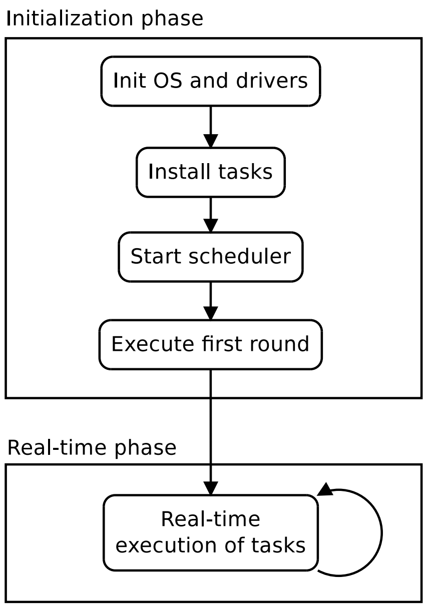


Figure 5.3: Execution rounds

5.3.1 Initialization Phase

The *initialization phase* is the non real-time phase during which the operating system initializes its data structures and creates the tasks with their associated message buffers, if they require IPC. Using a heap array from where the operating systems allocates memory for the tasks, the task control block (TCB), the tasks' message buffers and the stacks are created and initialized.

After all initializations have been performed the scheduler is started and the cyclic execution of the schedule begins. The first round is still executed in the context of the *initialization phase* during which the tasks' local variables are created ,put on the stack and the infinite loop is entered This is succeeded by an immediate release of control by a context switch to the *idle* task. After the schedule has been completely executed once all tasks are in an interrupted state just prior to the return from the context switch function, the *real-time phase* is entered.

5.3.2 Real-Time Phase

The *real-time phase* is started after the operating system, the hardware drivers and all tasks have been initialized. During the *real-time phase* tasks are not allowed to allocate further memory. Every task has to be executed as an infinite loop which gets preempted every time its execution for a given period is finished (or the task gives up execution voluntarily). Tasks can either run to completion by allowing it to execute until they perform the context switch to the *idle* task or can be preempted by the scheduler. Deadlines must be met under all circumstances.

5.4 Scheduler

The operating system uses a static table-driven, time-triggered preemptive scheduler as described in Chapter 2. As a result of the strict temporal control over task activations this approach guarantees good temporal predictability. The scheduler is only activated at predefined times and its invocation does not depend on the state of the tasks, as might be the case with C-tasks where blocking statements make scheduler invocation necessary. Its simple run-time decisions and low run-time overhead makes it a very well suited approach for keeping execution time jitter low.

The scheduler consists of a pre-run-time generated scheduling table that contains all the information needed to determine which task has to be started at what point in time. This table has to be created in a way that satisfies all precedence and mutual exclusion requirements. Hence the scheduler itself does not need to perform feasibility checks, which keeps the run-time overhead low. Not having to make any complicated run-time decisions, like determining the highest priority tasks, nor performing feasibility checks are the key benefits of this time-triggered approach. These ensure the temporal predictability at the system level as well as having a low number of possible decisions during execution. This small number of potential decisions keeps the number of branches to a minimum and enables the functions to have a constant execution time.

The scheduler is invoked every time an interrupt from the timer unit occurs and moves to the next entry in the scheduling table. It loads the new timer value from the table and performs a context switch to the next task. First all the values stored in the currently executing task's register are stored on the task's stack. Afterwards the register values of the next task are fetched from its stack, restoring the context of its last execution, giving the task control over the CPU back.

One *execution round* has been performed once the scheduler has completely moved through the whole scheduling table once and is ready to start the task at the first position in the table.

Resulting from the unpredictable timing of the hardware and the unknown characteristics of the application code, every task's first operation in the infinite loop has to be a context switch to the *idle* task. This is required to prevent the task from executing instructions which are supposed to be executed in the next run prematurely, in the case that the execution time is lower than the specified WCET.

If a multi node configuration is used the scheduling tables of all nodes have to share the following common structure:

- Each entry in the table must contain the same time value to prevent them from executing out of sync. A less strict requirement the sum of all time values of all tasks executed between input/output tasks has to be equal.
- whenever one task executes a *ttio_send_msg* task the corresponding task on the other node has to execute a *ttio_receive_msg* task. The hardware module's buffer is too small to store a useful amount of data for later reception.

5.5 Inter-process Communication

Task communication is implemented based on state-message semantics and uses a shared memory mechanism which supports 1-to-n non-consuming data exchange. As a result of using the state-message approach, old messages are completely superseded by newer versions, old messages do not have to be kept. This way a receiving task only needs to read the last message it received from any task that sends to it. If the communication were based on event-messages it would be required to read all messages a sending task has sent.

To prevent conflicting read/write accesses a circular buffer is used for each message. A situation with a writer using a circular buffer of size 9, sending a message for the third time in this execution round is described in Figure 5.4. Each time it writes a message it moves to the next position in the buffer. Reading tasks always follow-up and only access the buffer that was valid at the time of the reading tasks first access to the message. If the reading task is preempted by a sending task it will continue to read the old message even if the sender generated a new one. This is required to prevent conflicting read/write accesses, while still providing mechanisms with a constant execution times. The details of this mechanism are discussed in Section 6.4.

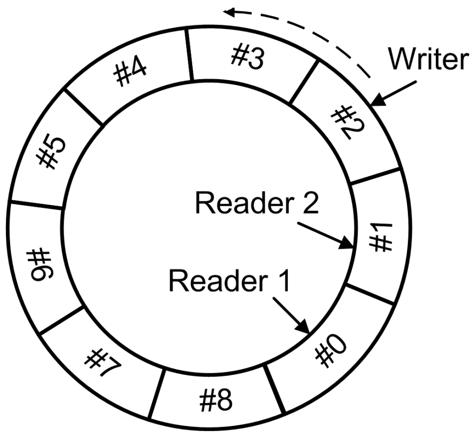


Figure 5.4: Circular buffer for a message to prevent overlapping read/write accesses

A global memory area, called message base, contains the start addresses of those buffers and the size of the message that has been stored there. Reading tasks can access the message base and fetch the pointer to the current buffer and directly access this memory area.

Two mechanisms to post the pointer to those buffers in the message base can be used. One is to store the pointer once the content of a message has completely been written to the buffer, or before the end of the task's current execution round. This has a very low overhead but the points in time when messages are made available for the reading tasks is not known.

The other method is to use an *ipc_send_msg* task that is scheduled with all the other tasks. The sending task stores the pointer in an intermediary message base which only an *ipc_send_msg* task reads. This task is invoked by the scheduler at predefined times. It reads the pointer in the intermediary message base and copies it to the permanent message base. All reading tasks can subsequently access the new message. The benefit of this approach is that the points in time when a message is made available to the reading tasks are known a-priori. The drawback is that it incurs a higher overhead than the other method.

5.6 Input/Output

For communication between different nodes a time-triggered I/O module is implemented. This consists of a globally accessible memory area called the I/O message base and two tasks used to access the communication medium at predefined times. The I/O message base directly contains the buffers used to store the message for delivery over the communication medium, respectively for the receiving task to read a received message. Normal tasks can read and write the buffers, indexed by I/O message IDs, in the I/O message base. This mechanism implements a temporal firewall as described in [Kop97]. The access to these buffers can happen at any time during the execution of the tasks. Special tasks are used to access the communication medium and send/receive the messages by accessing the I/O message base.

Figure 5.5 shows a sketch of this mechanism. The tasks can write into the I/O message buffers in the I/O message base at any time, indicated by the lines consisting of alternating dots and dashes, but the messages are exchanged only at predetermined times by the I/O tasks, indicated by the solid line.

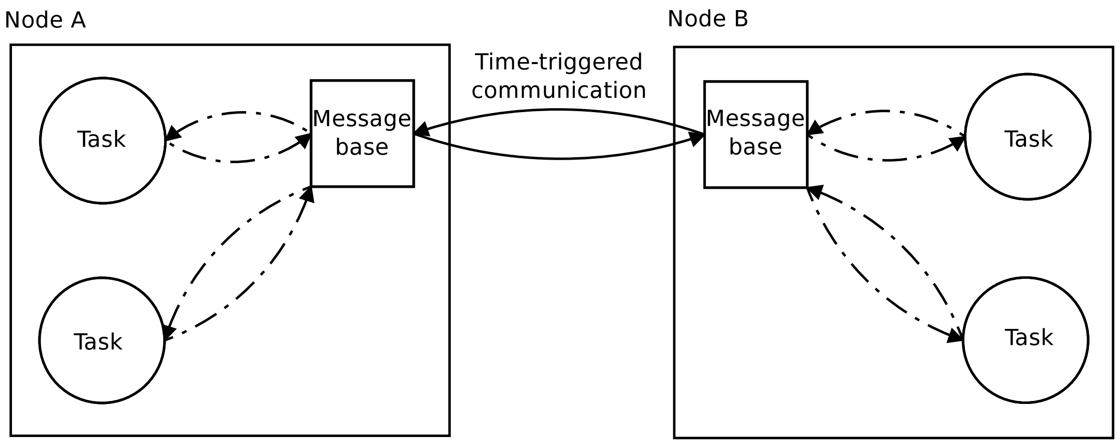


Figure 5.5: Time-triggered communication

CHAPTER 6

Implementation of the Operating System

Introduction

In this chapter we are going to discuss the implementation of the operating system. All the remaining modules are described and the details needed for application development are presented. We discuss the implementation and structure of tasks, the scheduler, IPC mechanisms and I/O mechanisms. The chapter is concluded with a description of how the operating system can be configured and envisioned extensions are mentioned.

6.1 Coding Style

The operating system is written using the WCET-oriented programming styles presented in Chapter 2. All loops are bounded by constants and there are no break or return statements in branching code. Also input data is never analysed in order to perform specialized functions that can improve the execution time in certain cases. As noted before, the single path approach cannot be used due to the missing conditional move instruction. The idea of serialising the execution as much as possible is applied in cases where execution time jitter could not be avoided otherwise. Whenever the execution time depends on certain properties, like the maximum size of a message, these are defined pre-run-time and therefore do not result in varying execution times during the execution of the system, but for the analysis these variations have to be considered. Branches that could not be avoided are written in a way that equalises the execution times of the alternatives as much as possible. This was done following a similar approach as the *if-translation* of the single path approach. Also whenever dynamic data structures, like linked lists, would be used in traditional programming, arrays of predefined size are used as lookup tables indexed by the associated IDs. This trades memory efficiency for temporal predictability but is essential to keep the execution time constant and not just boundable.

6.2 Task Creation

During the *initialization phase* of the system tasks are installed by calling the *task_create()* function. The following parameters have to be passed to this function:

- Function pointer to the tasks function
- A task name
- A unique taskID to identify the task
- The size of the stack in elements (this should be of the type double)
- A typeID that states what type of task it is (*TASK*, *TTSEND*, *TTRECEIVE*, *WAIT-SYNC* and *IPCSEND*)

The *task_create()* function allocates memory for the task control block (TCB) and initializes it. The TCB structure is summarized in Table 6.1. This structure contains the stack pointer, as well as pointers to the start/end of the stack and a variable indicating the stack size. It also contains an array of size *MAX_MSG* that contains pointers to the position in the circular buffers (of each message). If a send operation takes pace it writes to the buffers these pointers point to. In order to link tasks to the schedule a TaskID field is present and a task name can be assigned. Using the taskID the scheduler (*parse_schedule()*) can link the entry of the scheduling table, which is implemented as an array, to the task that is being created and store the pointer to the TCB in this scheduling array.

As a last step in the task creation the stack is initialized by calling the *port_initialise_stack()* function. This function depends on the hardware architecture and has to be adapted accordingly if the operating system is ported to a different architecture.

Before the function returns it calls the task's initialization function that can be used by the programmer to create optional message buffers for IPC. If no IPC is required the body of that function can be empty, but this function has to be present nevertheless. The programmer is free to use this function for other initializations.

The *main()* function, used as the entry point of the system's execution, has to consist of the call to the *OS_init()* function as well as install all necessary tasks. If the scheduler is started by an event rather than manually in the *main()* function, an infinite loop preventing the exit of the system is required.

Field Name	Description
*TopOfStack	Pointer to the location of the last item placed on the task's stack
*Stack	Pointer to the start of the stack
*EndStack	Pointer to the end of the stack
TaskID	ID of the task
StackSize	max size of the stack
*msgBufferArray[MAX_MSG]	Array holding the pointers to the buffers of each message
TaskName[MAX_TASK_NAME_LEN]	Descriptive name given to the task when created.

Table 6.1: Task Control Block

6.3 Scheduler

The predictable temporal behaviour of the time-triggered static table-driven scheduler makes it the ideal choice for this operating system. It provides the capabilities to have a constant execution time due to the simple scheduling decision during run-time.

The scheduler cyclically reads the scheduling array which has to be generated before the execution of the system. This array specifies the points in time the tasks have to be started. The fields of the structure are summarized in Table 6.2. The *TCBPtr* field is used to link the tasks to the array positions. This value is only known at run-time. To locate the task in the scheduling array a *taskID* is needed. The *iomsgID* and *ipcmsgID* are required if the tasks are *I/O* tasks, respectively the *ipc_send_msg* task. To determine the type of the task the field *type* is used. Also to make it easier for the developer to identify tasks, a *task name* can be stored. The decision which task to execute next consists simply of a fetch of the next task's stack pointer from where the task's context is restored. Afterwards a counter variable, that indicates the position of the task to switch to next, is incremented, or set to 0 if the end of the array has been reached. These operations are completely independent of the type of the tasks, the state the task is in and the number of tasks.

On architectures that do not support conditional move instructions (like the SPARC v8), or use unpredictable hardware features, jitter in the task execution time can not be ruled out completely. Therefore it is not always possible to specify a timer value that interrupts the execution of the task directly after its last instruction has been executed. The execution round of a task is equal to one iteration of the infinite loop. A timer value that is larger than the tasks execution time allows the task to start executing instructions belonging to the next round. The task then could generate data that is outdated by the time it is actually allowed to execute, which is of course undesirable.

To prevent this glitch a *task_finish()* function call has to be placed at the beginning of the tasks infinite loop. Depending on the operating system configuration this function performs a context switch to an *idle* task, preventing the task from executing the next round, or invokes the scheduler that starts the next task. The *idle* task is nothing more but an infinite loop without containing any instructions. The time slot the scheduler assigns to each task in a given round must be at least as long as the task's WCET plus the time needed to perform the context switch to the *idle* task. If a task is preempted, and therefore executes more often than once in a given round, the sum of all it's execution time must be equal to it's WCET plus the context switch to the *idle* task.

Due to the dependence of the timer's decrementor on the number of implemented timers one timer tick can only occur after as many clock ticks as the number of implemented timers. This means that, if for example, 4 timers are implemented a timer tick can only occur every 4th clock cycles. If more than one timer is used this overhead has to be considered, when specifying the timer values as well.

The *task_finish()* functions implementation differs from the scheduler's only in that it does not set the timer to a new value and always performs the context switch to the *idle* task.

The operating system currently supports 3 possible task execution modes:

- Cooperative: The scheduler is not used in a time-triggered fashion by not enabling the timer interrupt. Every task runs until it calls the *task_finish()* function which invokes the scheduler.

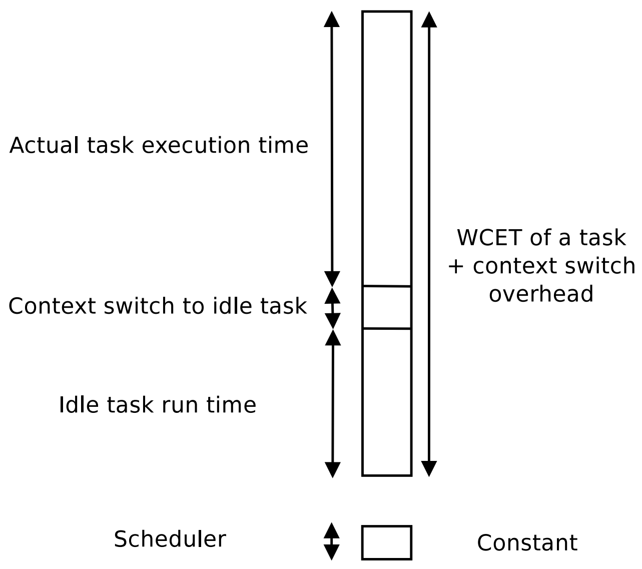


Figure 6.1: Composition of the timer value

- Preemptive: Tasks run until they are preempted by the scheduler. On architectures where it is possible to have a constant execution time the scheduler is started exactly after the last instruction of the task (not possible on SPARC v8).

- Combination: On architectures where the execution time is not constant a combination of both is needed. A task relinquishes control once it is done by calling the *task_finish()* function. This prevents entering of the next loop iteration prematurely. Also tasks can be preempted before that call as long as they are scheduled to finish their execution later in the same execution round.

6.4 Inter-process Communication

The task communication is implemented based on state-message semantics outlined in Chapter 5. It supports a 1-to-n non-consuming data sharing based on shared memory mechanisms.

To communicate every sender has to create a circular buffer containing ≥ 1 buffers of size *MAX_IPC_MSG_SIZE* associated with each message. The number of buffers depends on how often this task sends a message under the same ID within one execution round.

Field Name	Description
*TCBPtr	A pointer to the TCB, this is added by the parse_schedule() function
taskID	The ID of the task
time	The timer value indicating the time when the next task is allowed to start, this consists of the WCET of the current task + the overhead for the context switch to the *idle* task
iomsgID	If the task is a *ttio_send_msg* or *ttio_receive_msg* task we need to tell it which I/O message to process
ipcmsgID	If the task is an *ipc_send_msg* task we need to tell it which IPC message to process
type	The type of the task, possible values are: *TASK*, *WAITSYNC*, *TTSEND*, *TTRECEIVE*, *IPCSEND*
taskName[]	The name of the task

Table 6.2: Schedule table entry

A global memory area, called message base, contains the start addresses of the most recently filled buffer of each message and the size of the message. Which buffer should be accessed in the current execution round is being tracked in the task's Task Control Block (TCB).

Every task that needs to send messages has to call the *msg_create()* function in the task's initialization function. This function requires a message ID, the maximum size of the message in bytes and the number of buffers as parameter. This function allocates the required buffers and initializes the values in the message base. In case preemption is allowed a sender must switch to a different buffer when the write access can overlap with the data being read by a receiving task to prevent conflicting accesses. A safe upper bound on the number of buffers needed to prevent such conflicts is the number of times the sending task executes in one *execution round*.

To fill the correct buffers the tasks call a *store_msg()* function which needs the MSG_ID, the size of the data element in bytes and the data element to be stored as parameters. This function's execution time depends on the *MAX_IPC_SIZE* constant that specifies the maximum size a message can have.

Another possibility to access the buffer is to use the *get_buffer_ptr()* function which returns the pointer to the current buffer for the specified msgID. This access method can be useful if it is possible to store the data in this buffer without requiring to copy it there. This can for example be the case if the data is contained in a structure. Overlaying the buffer with this structure allows to write the data directly, eliminating the overhead of the copy mechanism.

There are two mechanisms to make messages available to the reading tasks. The first method is by calling the *send_msg()* function which posts the pointer to the current buffer to the message base for the other tasks to see. This function also prepares for the next send operation by moving to the next position in the cyclic buffer. The sending task will then write to a different buffer the next time it stores a message. If only one buffer is needed, due to the sending task executing only once per round, this function sets the pointer to the same address again. This superfluous action is needed to keep the execution time of all cases constant, or otherwise the code executed in this function would depend on the number of buffers and therefore different execution paths would be present.

The other mechanism is to use a special task to store the pointer in the message base. If this IPC send mechanism is used the write-action done by the *send_msg()* function is performed on an intermediary message base and the *ipc_send_msg* task takes the pointer from the intermediary message base and stores it the pointer in the global message base when it is started by the scheduler. This approach allows the precise control over the instants of time a message is made available to reading tasks. It is therefore easier to determine if a receiving task is operating on outdated data. This information is not available when the other method is used owing to the lack of knowing whether a task is preempted before or after executing a *send_msg()* call.

Figure 6.4 shows the possible methods for accessing and sending of IPC messages. The full ellipses indicate operations of operating system functions whilst broken ellipses indicate actions performed by application code.

A task that requires to access a specific message does so by first executing a *get_msg()* function call which returns the pointer to the current buffer from the message base. Following this the task can access the buffer without any restrictions. Since no data protection mechanisms are available and every task can access the data, the programmer is not allowed to write into this data segment.

The next time a task wants to store a new message under the same message ID it will simply write into a different buffer. No reading task that has already accessed the buffer has any knowledge about this other buffer, which prevents conflicting accesses. Data consistency has to be guaranteed at compile time by specifying a sufficiently large number of buffers for each message. The IPC module does not impose a particular structure for the messages. Instead the programmer has complete control over this.

The msgBuffer structure used in the message base is described in Table 6.3. It contains an ID to identify the buffer (field *msgID*), the number of buffers (field *NumberofBuffers*) required for the circular buffer, a pointer to the buffer most recently written to (*MsgBuffer-ArrayPtr*), the size of the message in this buffer (field *curSize*) as well as the max size of a message (field *maxSize*) and the position in the circular buffer indicated by the field *cur-Buff*. During a *send_msg()* call the value of *curSize* and *MsgBufferArrayPtr* in the TCB is stored in the message base. The message base is an array of the structure described in Table 6.4 and contains the pointer to the msgBuffer (field *msgBufferPtr*) and the size of the message stored there (field *curSize*). The array is indexed by the msgIDs and its size depends on the *MAX_MSG* constant.

The execution times of all IPC related functions are constant either as a result of using bounded loops and the lookup tables in the TCB and the message base.

Field name	Description
msgID	The ID of this message
NumberofBuffers	The total number of buffers for this message
*MsgBufferArrayPtr	The pointer to the current buffer
curSize	The size of a message currently stored in the buffer
maxSize	The maximum size of a buffer
curBuff	The number of the buffer currently in use

Table 6.3: Structure controlling a message buffer

Field name	Description
curSize	The size of the message currently stored in the buffer
*msgBufferPtr	Pointer to the currently used msgbuffer

Table 6.4: Message base entry

We will now discuss the sequence of actions taken by the system to successfully transmit messages. Before the first message is sent the buffer pointer in the TCB structure points to the start of the first buffer while the pointer in the message base is still null, as shown in Figure 6.2. Figure 6.3a displays the state of the data structures after a first send operation has been executed. The pointer in the TCB moves to the next buffer, the sending task is therefore forced to write to a different memory location the next time it is executed. The old address is posted in the message base for the reading tasks.

Another send operation during a subsequent execution of the same task again moves the pointers to the next buffers and so forth. This can be seen in Figure 6.3b. If the correct

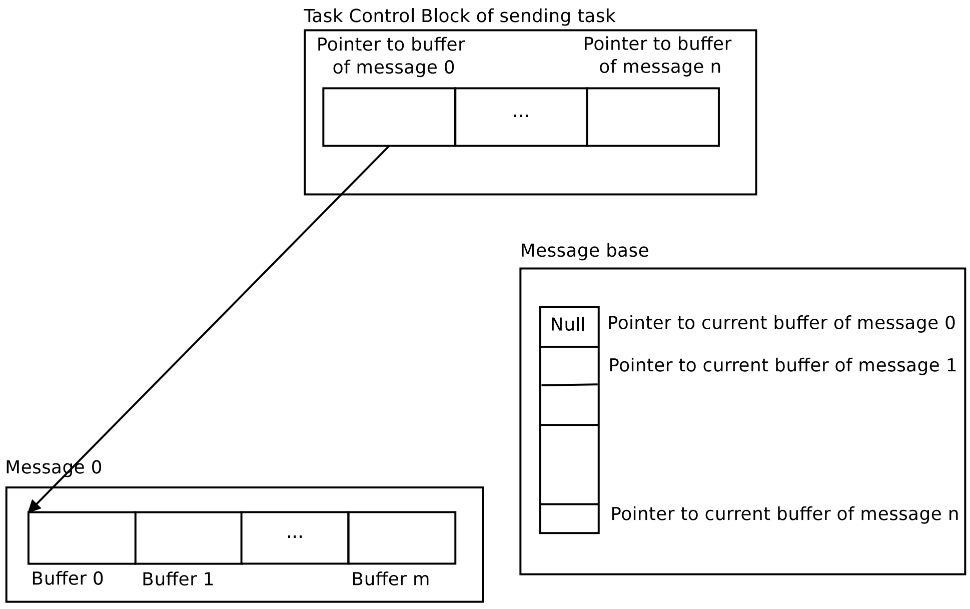

Task Control Block of sending task
Pointer to buffer of message 0
Pointer to buffer of message n
...
Message base
Null
Pointer to current buffer of message 0
Pointer to current buffer of message 1
Pointer to current buffer of message n
Message 0
...
Buffer 0
Buffer 1
Buffer m

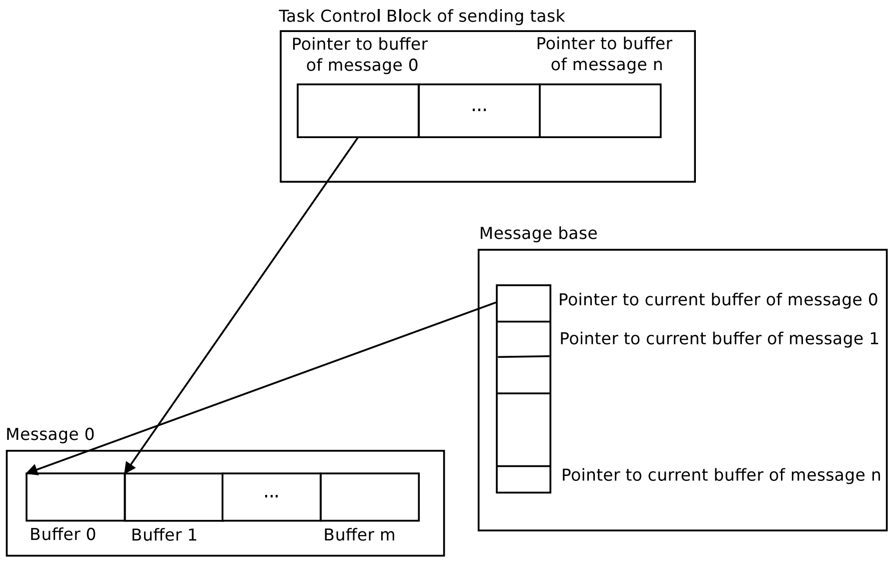
Figure 6.2: State of the data structures before the first send operation

number of buffers is used the pointer will again reference the first pointer at the beginning of the next execution round.

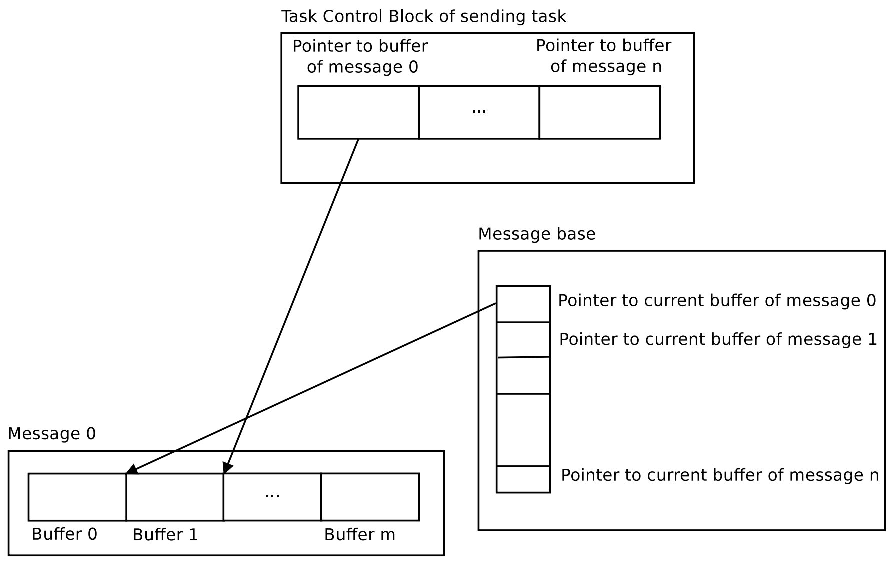
This is required to prevent a reader/writer access problem when a writing task and reading task can be interleaved. Non blocking algorithms dealing with such a situation were described in Chapter 2. The solution implemented here is similar in concept to the Double Buffering Algorithm [HS02], using a circular buffer that is used to store all the messages generated for the same message ID consecutively.

Let us now discuss the requirement for the circular buffer. Imagine that a sending task always writes to the same buffer. The task can not be allowed to run more often than once during an execution round or otherwise the preemption of a reading task by the sending task could result in data corruption. This problem is demonstrated in Table 6.5.

A sending task writes the message ”1234“ during its first execution. While the reading task accesses the buffer it is preempted and has read only the partial message ”12“. At a later time but before the reader is allowed to continue to execute the sending task is executed again and writes the new message ”5678“. Eventually the reading task is invoked and continues reading but now reads the partial message ”78“ resulting in reading the incorrect message ”1278“ instead of the correct old message ”1234“ or the newer version ”5678“. By using the circular buffer the reader will only read the message that was valid


Task Control Block of sending task
Pointer to buffer of message 0
Pointer to buffer of message n
...
Message base
Pointer to current buffer of message 0
Pointer to current buffer of message 1
Pointer to current buffer of message n
Message 0
...
Buffer 0
Buffer 1
Buffer m

(a) State of the data structures after the first send operation


Task Control Block of sending task
Pointer to buffer of message 0
Pointer to buffer of message n
...
Message base
Pointer to current buffer of message 0
Pointer to current buffer of message 1
Pointer to current buffer of message n
Message 0
...
Buffer 0
Buffer 1
Buffer m

(b) State of the data structures after the second send operation

Figure 6.3: Visualization of send operations

at the time it started accessing. During this read a sender might write a new message but does not interrupt the reader by using a different buffer.

This problem could also be averted by switching to other message IDs every time the accessing task sends a message under a previously used ID. This has the drawback that it would require implementation work by the programmer or tool support. But this effectively implements the same mechanism described above in user code and hence is not advisable.

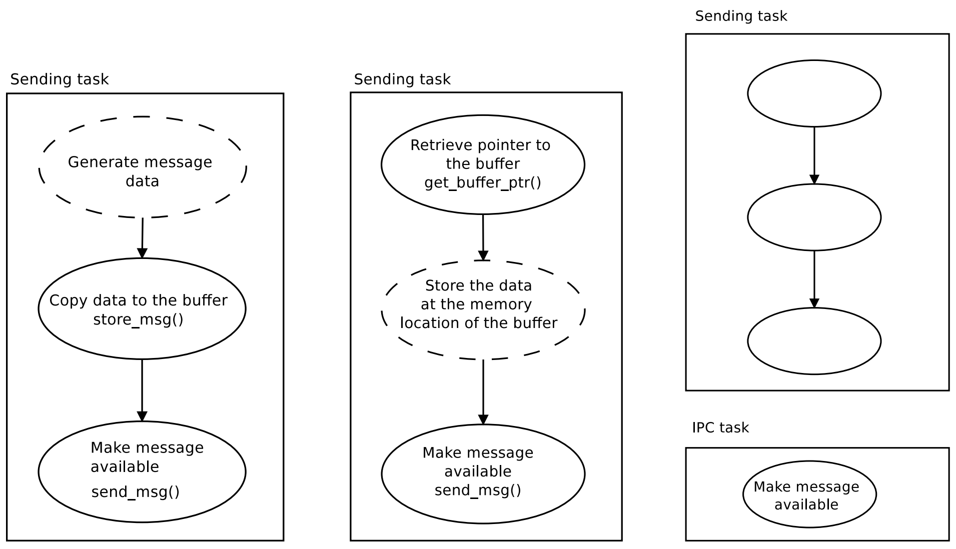
Time	Sending task		Buffer	Reading task		Event
	Store value	Element #		Value read	Element #	
1	1	1	1		1	
2	2	2	12		1	
3	3	3	123		1	
4	4	4	1234		1	
5		4	1234		1	send done
k		1	1234		1	read task start
k+1		1	1234	1	1	
k+2		1	1234	12	2	
k+3		1	1234	12	2	task preempted
j		1	1234	12	2	send task start
j+1	5	1	5234	12	2	
j+2	6	2	5634	12	2	
j+3	7	3	5674	12	2	
j+4	8	4	5678	12	2	
j+5		4	5678	12	2	send done
h		1	5678	12	2	read task resume
h+1		1	5678	127	3	
h+2		1	5678	1278	4	

Table 6.5: Access problem without multiple buffers

6.5 Input/Output

For Input/Output operations a simulated time-triggered I/O module is implemented because no hardware module transparently dealing with data transfer between nodes was available for this platform. Tasks can read and write a buffer called the I/O message base which implements a temporal firewall [Kop97]. The I/O message base is a globally accessible memory area that contains buffers for all I/O messages. Figure 5.5 showed this setup. The tasks write into the buffers and the messages are exchanged at predetermined times by scheduled I/O tasks.

When using a hardware module data transfer between the host and the communication medium uses a dual ported RAM which is accessed by a hardware module to handle the


Sending task
Generate message data
Copy data to the buffer store_msg()
Make message available send_msg()
Sending task
Retrieve pointer to the buffer get_buffer_ptr()
Store the data at the memory location of the buffer
Make message available send_msg()
Sending task
IPC task
Make message available

(a) Copy the data to the buffer (b) Use direct memory access to the buffer (c) Additionally use an IPC task

Figure 6.4: 3 possible ways of IPC communication

time-triggered traffic. All access times to the communication medium as well as the RAM itself could be determined beforehand.

Since such a module is not available for the LEON3 CPU the operating system uses two special tasks called *ttio_send_msg* and *ttio_receive_msg* that are scheduled like any other task. These tasks read and write to the temporal firewall and communicate with other nodes via the UART hardware module. The programmer does not have to modify these tasks but only add them to the scheduling array and provide a timer value large enough to send a message of maximum size. This ensures that accesses to the communication medium occur only at predefined times.

The scheduling arrays used by the nodes must share a common structure. The invocation times of *ttio_send_msg* and *ttio_receive_msg* tasks have to be the same on every node and when one node executes a *ttio_send_msg* task the other has to execute a *ttio_receive_msg* task. This is required because the UART provides only a small buffer that is often not sufficient to hold all data that has to be exchanged.

Due to the unpredictable nature of the communication medium the execution time of these TT tasks cannot be constant.

To access these buffers the *write_io_message()* and *read_io_message()* functions are provided, which require a task local buffer.

6.6 Memory Management

A memory allocation function called *prvmalloc()* is used to allocate memory from a global array of constant size whose size has to be specified at compile time. The function is called with a parameter specifying the size of the requested memory block and returns a pointer to the start of such a block. The function keeps track of the next free element in the array. A dummy *free()* operation is provided for consistency but calling it has no effect. Figure 6.5 shows the implementation. The array has size m and a pointer is used to keep track of the sum of allocated elements. When memory is requested the pointer to the first free element is returned (element n) and is moved to the element with index $n + size$ where size is the size of the requested data block.

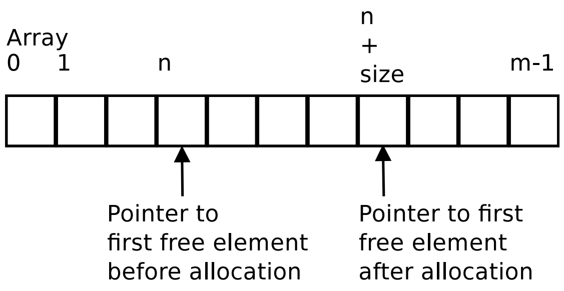

Array
0
1
n
n + size
m-1
Pointer to first free element before allocation
Pointer to first free element after allocation

Figure 6.5: Heap array and the pointer being returned to the caller

6.7 Drivers

Drivers for the following hardware features are provided:

- 2 7-Segment displays
- The IRQ module (IRQ 2 reserved for UART, IRQ8 reserved for the scheduler timer)
- Timers (Timer 0 used for the scheduler)
- LEDs

- Switch button
- The general purpose I/O module
- UART I/O

6.8 Extending the Operating System

6.8.1 Configuration of the Operating System

To adapt the operating system to different environments many parts of the operating system are configurable using pre-processor *defines*. This section describes the most important *defines*. The following *defines* have an impact on the WCET:

- *MAX_IO_MSG_SIZE*: This constant defines the size an I/O message can have at most. This has an impact on the loops in the *write_io_message()* and *read_io_message()* functions.
- *MAX_IPC_MSG_SIZE*: This constant defines the size an IPC message can have at most. This has an impact on the loop in the *store_msg()* function.

Defines adding additional features to the operating system:

- *SCHEDULE_IPC*: With this flag set IPC messages are only made available by a special *ipc_send_msg* task which has to be scheduled instead of being sent by the writing task.
- *TTIO_MODSIM*: This activates the a time-triggered I/O module. It adds a global buffer to write messages to and requires *ttio_send_msg* and *ttio_receive_msg* tasks to perform the communication.

The following *defines* change the functionality of the operating system:

- *TTSCHEDULER*: The scheduler is invoked by a timer interrupt and the *task_finish()* function always performs a context switch to the *idle* task.

- *COOPERATIVE*: The scheduler is invoked voluntarily by the *task_finish()* function call at the end of the task's execution round. (Note that using *TTSCHEDULER* and *COOPERATIVE* is mutually exclusive)
- *BOUNDED_LOOPS*: If this *define* is not present loops run depending on the data size instead of the provided constants. This makes WCET analysis impossible.

The following *defines* are used for debugging:

- *DEBUG*: enables debug code, must be combined with one of the following:
  - *PRINTF*: This uses the printf function provided by the stdio.h from the newlib. This can be used in single node mode when timing is not important to print to the UART.
  - *SEG*: This outputs error codes on the LEDs and the 7-Segment display.
- also each file provides a *DEBUG_%MODULENAME define* allowing to selectively enable debug code/output for the specific files.

Defines changing the amount of memory used by the operating system are:

- *HEAP_SIZE*: The total size of the array used by *prvmalloc()*. This can be the total size of the available memory.
- *MAX_IO_MSG*: The total number of possible I/O messages. This changes the size of the buffer in the I/O message base.
- *MAX_MSG*: The total number of possible IPC messages. This *define* is used to specify the size of the the message array in the TCB and the message base. See Table 6.3 and 6.1.

6.8.2 Extending the Operating System

Some extensions to the operating systems were already envisioned during the development and are described briefly in this section. In the current implementation a context switch to the *idle* task is performed when the task reaches the end of its loop iteration for the given execution. In case the execution time of the tasks are much shorter than the time provided by the timer value in the scheduler a large amount of computational resources is wasted. In some scenarios there may exist tasks that do not have real-time requirements, for these it would be enough to be executed sporadically. Therefore it could be beneficial to implement a small non real-time scheduler that is invoked instead of performing the context switch to the *idle* task. The non real-time tasks could use the remaining available time until the next real-time task is executed to increase the CPU utilization instead of executing the useless *idle* task. This scenario is shown in Figure 6.6.

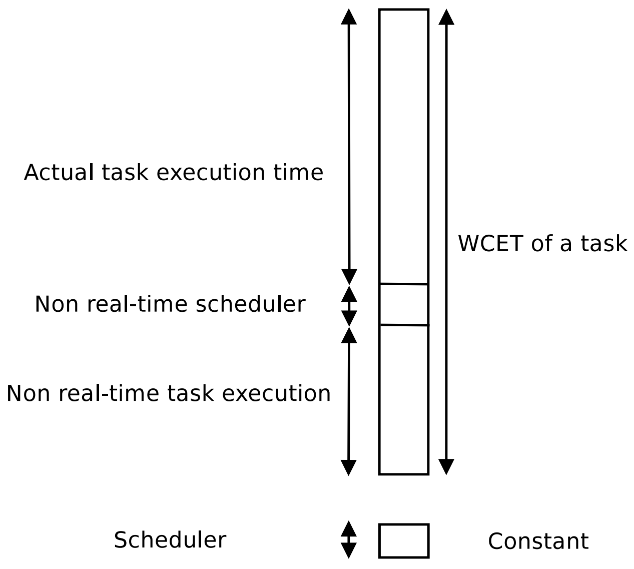


Figure 6.6: Time composition using an additional non real-time scheduler

Another possible extension is to use a hardware time-triggered communication module. Many such implementations use a dual ported RAM as the interface between the host and the module. In the case that this is mapped into the address space of the CPU it would be sufficient to overlay this memory location with the temporal firewall structure. If a specific message structure is required by such a module changes to the *read_io_message()* and *write_io_message()* functions would be required as well.

The IPC message creation function currently requires the programmer to specify the number of buffers needed to ensure data consistency. This could be automated by counting how often a task is found when the scheduling array is parsed. The operating system could then transparently generate the required amount of buffers . This would however increase the memory overhead whenever a lower number of buffers will be sufficient, but it would make the message creation less error prone.

CHAPTER 7

Evaluation

Introduction

On the following pages we are going to discuss the goal of the evaluation and how it was carried out. As a first step we present the applications that were used for testing and measurements. Following that the measurement results of all operating system functions, together with the various configurations, are presented and discussed. As a last step the results are summarized and a short conclusion is given.

7.1 Aim of the Evaluation

As a first step of the evaluation the functional correctness of the system had to be verified. For this a small demo application utilizing all modules at once was developed.

As a second step the WCET analysis tool aiT was used to locate branches that could impact the execution time. Test data for these cases was then derived accordingly. As a secondary goal it was also determined how easy it was to apply such tool to the operating system code.

The concepts for IPC and scheduling guarantee predictable temporal behaviour as long as the execution times of the applications are known. Owing to this it is essential that all function the operating system provides have a constant execution time, or at least experience only a small execution time jitter.

To determine whether all operating system functions really have constant execution times, and to determine the jitter if it can not be avoided, measurements were carried out. This required to generate test data that covers all the possible variations that might have an impact on the execution time, like the system state and the function parameters. If jitter was found the code was revised trying to remove the jitter if possible, and the measurements were repeated.

7.2 Test Application

To test the functional correctness of the operating system a two-node client/server setup was used.

An interrupt triggered by a button press on the server activates an output of a GPIO pin, This itself triggers an interrupt on the client. These interrupts start the execution of the schedulers on both nodes. The first task execute was a *wait_sync* task. This task synchronizes the client's clock to the server's with enough precision to allow the exchange of messages. Each node executes three tasks together with *ttio_send_msg* and *ttio_receive_msg* tasks, an *ipc_send_msg* task and the before-mentioned *wait_sync* task.

The server node executes the following computations:

- Task 1 executes a *write_io_message()*, writing a hexadecimal value as a char to the I/O message with ID 2, and increments the value.
- Task 2 does nothing but set LEDs to indicate that it was executed.
- The tasks *ttio_send_msg* and *ttio_receive_msg* are executed to send I/O message 2 and receive I/O message 1.
- Task 3 reads the I/O message with ID 1 from the I/O message base and displays the value it received on the 7-segment display.

The client node concurrently executes the following computations:

- Task 1 communicates with task 2 via the IPC message with ID 0 that contains a hexadecimal value which it incremented after it has been sent.
- Task 2 reads the message with ID 0 and stores the value in the I/O message base with the I/O message ID 1.

- The tasks *ttio_receive_msg* and *ttio_send_msg* are executed to receive I/O message 2 and send I/O message 1.
- Task 3 reads the I/O message with ID 2 from the I/O message base and displays the value it has received on the 7-Segment display.

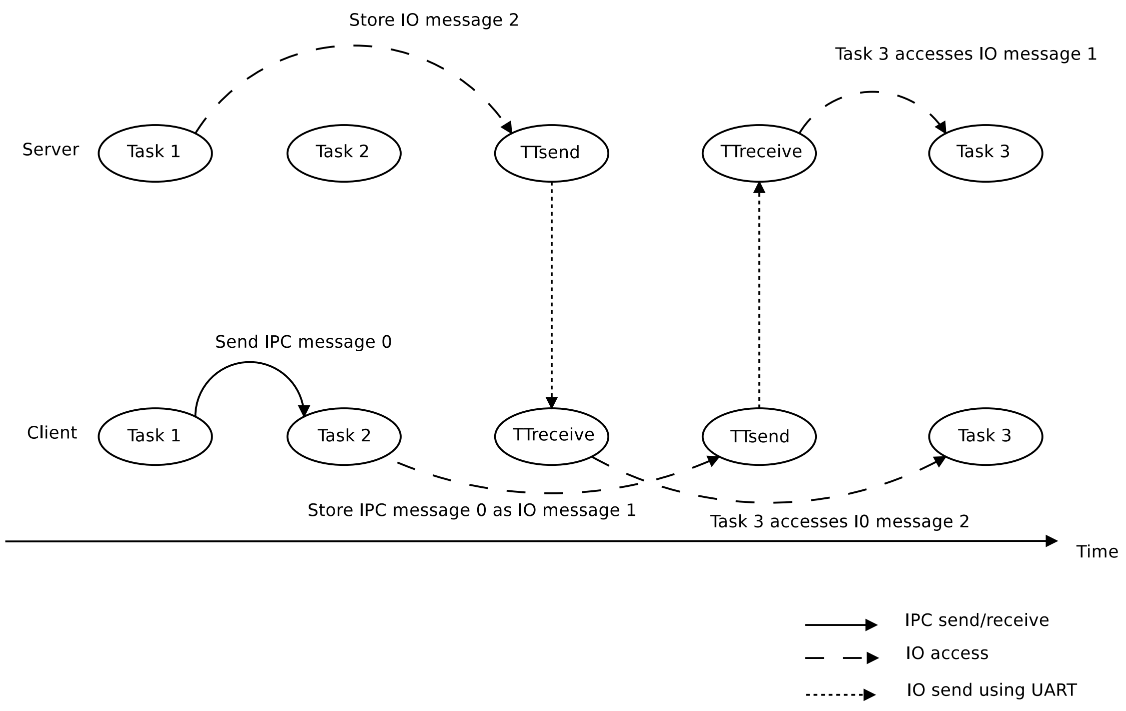


Figure 7.1: Testing application

7.3 Static Analysis

The tool aiT was used to identify branches that could lead to execution time jitter. Using the control flow graphs the tool generated test data was generated for the identified branches.

Apart from assisting in the generation test data the tool also found a potentially infinite recursion that could occur if an error is caught and the 7-Segment display used to display the error ID is not available.

A short experiment was performed trying to analyse the temporal behaviour of the operating system with this tool. It was able to analyse all functions called during the *real-time* *phase* was able to find upper bounds for all loops. Some flow annotations were required to inform the tool of branches that could not be entered in the same execution trace.

Unfortunately the hardware model used for the LEON3 core does not seem to be sufficiently accurate, in that the results obtained differed a lot from the measurement results. This tool calculated a nearly 2 times higher WCET for the *write_io_message()* function than what the subsequent measurements revealed. The tool offers options to configure the hardware model like deactivate caches and set the latency for the memory accesses. Although the tool was configured not to use caches and that memory accesses have 0 cycle latency different settings for the cache behaviour still resulted in different results. The confidence in the correctness of the results is therefore rather low. The tool also encountered paths it sometimes, depending on the pipeline analysis selected, considered to be impossible to analyse or unbounded.

7.4 Measurement Application

For the measurements a single-node setup was used on account of being able to use the debugger for accessing the timer value register.

The application developed for the measurement has the following structure: Task 1 is used to test the different possible IPC send mechanisms. It writes 6 different IPC messages:

- IPC message IDs 0 and 1 are used to store strings of different lengths that are stored in local variables. This verifies that the size of the message has no impact on the execution time.
- The pointers to the buffers associated with message ID 2 and 3 are retrieved by a *get_buffer_pointer()* call and the same strings used for message IDs 0 and 1 are written to the memory directly. This verified that the *send_msg()* function does not depend on the way the message content has been stored.
- The pointer for message ID 4 are used to write a structure, containing a string array and two integers, directly to the buffer. This verified that the *send_msg()* function does not depend on the type of the message content.
- The same structure used for ID 4 is stored in the buffer for message ID 5 but this time written by a *store_msg()* call. This verified that the *send_msg()* function does also not depend on this combination of message content and way of storing it.

Task 2 is used to access the messages sent by task 1 using the *get_msg()* function. It also writes 2 strings of different lengths, that are stored as local variables, to the I/O message base. Finally task 3 is used to read the I/O messages “sent” by task 2.

This setup allowed to measure of all IPC and scheduler functions as well as the access to the I/O message base. The code for this application can be found in Appendix B.

7.4.1 Test Setup

To measure the execution times of all the function calls executed during the real-time phase, the execution time (in cycles) was measured using the time tag counter provided by the DSU3 Leon3 Hardware Debug Support Unit. This counter is a cycle-accurate 30 bit counter that is incremented each time an instruction is executed. Measurements were performed using the debug facilities provided by the LEON3 IDE plug-in for Eclipse. Breakpoints were set before and right after the function call of the function under measurement. Due to this not only the execution of the function, but also the passing of the parameters were included in the measurement. At each breakpoint the value of the time tag counter was read from its memory location at 0x90000008 and the difference between the values read at the breakpoint before and after the function call was calculated. The LEON3 CPU executes a trap instruction each time a breakpoint is hit, adding a constant 5 cycles overhead to each measurement.

Each measurement assessed the execution time depending on the parameters that could have an influence on the execution time of the function. This constitutes a white box testing methodology in that the knowledge about the implementation was used to derive test cases.

The LEON3 core used for these measurements uses dedicated instruction and data RAMs synthesized into the FPGA. These provide constant 0 cycle access. No caches, branch prediction hardware or external SDRAM were used due their dynamic timing that would prevent us from determining whether the code or the hardware was responsible for jitter.

All tests were performed without optimizations unless noted otherwise. This ensured that the code executed is actually as it was intended and prevent optimizations, that might not be beneficial for the worst case and therefore introduce jitter from being carried out.

All functions were measured over four real-time execution rounds. This was sufficient in that it allowed to observe every possible parameter that could impact the execution time and the data generated in each round was the same. The only function that could experience different execution rounds depending on how often it is performed is the buffer switching function. Usage of messages with different circular buffer sizes allowed to observe all cases of buffer handling.

7.5 Scheduler

The different scheduler actions that were measured are:

- The execution time of the scheduler plus the context switch without the scheduled IPC mechanism (*IPC_SCHEDULE* undefined).
- The execution time of the scheduler plus the context switch with the scheduled IPC mechanism (*IPC_SCHEDULE* undefined).
- The execution of *task_finish()* when using the time-triggered scheduler (*TTSCHEDULER* defined and *COOPERATIVE* undefined).
- *task_finish()* invoking the scheduler (*TTSCHEDULER* undefined and *COOPERATIVE* defined).

7.5.1 Results of the Context Switch

As a result of always allowing the tasks to run to completion this breakpoint was set directly at the position in the *task_finish()* where the task was previously interrupted. The start point of the measurements was set to the entry of the interrupt service routine. It therefore also measured not only the scheduler function that determines the next task, but also the execution time of the interrupt handler code provided by the newlib library.

Context Switch Without SCHEDULE_IPC

The execution of the context switch was measured using a scheduling array containing 4 entries (executing the *wait_sync* task and 3 user tasks). Two execution rounds using this scheduling array showed a constant execution time of 990 cycles.

Context Switch With SCHEDULE_IPC

Execution of the context switch was measured for a schedule array containing 5 entries (executing the *wait_sync* task, the *ipc_send_msg* task and 3 user tasks). Two execution rounds using this scheduling array showed an execution time of 1029 cycles. The difference to the execution without the *ipc_send_msg* task (*IPC_SCHEDULE* undefined) originated from the removal of code needed only when using the scheduled ipc send mechanism (not defining *IPC_SCHEDULE* removes that superfluous piece of code).

Context Switch With More Tasks and SCHEDULE_IPC

To show that the execution time does not depend on the number of tasks another measurement was performed with a scheduling array containing 8 tasks (1 *wait_sync* task, 2 *schedule_ipc* tasks and 5 user tasks). Again the execution time was constant at 1029 cycles.

Further Context Switch Tests

Another measurement was performed without using the scheduled IPC send mechanisms (*IPC_SCHEDULE* undefined) but with retaining the code piece that was removed in the "context switch without *IPC_SCHEDULE*" test, that resulted in this 39 cycle difference. This code has no effect in this configuration and can be executed regardlessly. This time the execution was again 1029 instead of 990 cycles long. This version was also tested with O2 optimizations enabled. This then experienced an execution time between 770 and 773 cycles, depending on which task to switch to next. This jitter was constant in all observed execution rounds.

7.5.2 Results of the task_finish() Function

task_finish() With the Time-triggered Scheduler - TTSCHEDULER MODE

Measurements for the *task_finish()* functions were taken from before the execution of the function to the infinite loop in the *idle* task before an $i++$ instruction. This instruction was needed owing to the fact that without it the debugger would crash at the breakpoint. The measurements experienced a jitter of 5 cycles due to the 7 cycle length of the infinite loop. This is due to the lack of knowledge the point in the 4 instruction long execution trace, the execution is interrupted. The code that lead to this jitter is shown in Table 7.4. In the worst case the task is interrupted just after the inc instruction which means that there is a 7 cycle delay until the breakpoint is reached again. In the best case however the task is interrupted before the ld instruction which means only a 2 cycle delay is experienced.

The results of 4 executions of the 4 tasks containing scheduling array (starting with the first real-time round) showed an execution time of 768 to 773 cycles.

task_finish() Without the Time-triggered Scheduler - COOPERATIVE MODE

If we do not use the scheduler the call of *task_finish()* switches to the next task in the scheduling table. This was measured from before the call of *task_finish()* to before the first instruction in the next task. This was measured over 4 executions of the scheduling table beginning with the first real-time round and showed a constant execution time of 781 cycles.

7.6 IPC

The following IPC functions were tested:

- *store_msg()*
- *send_msg()*
- *get_buffer_ptr()*
- *get_msg()*
- *ipc_send_msg* task

The impact of the parameters: message ID, size of the message and number of buffers was evaluated.

7.6.1 Results of the store_msg() Function

Three measurements for *store_msg(int msgID, void* sourcePtr)* were performed, 2 used a constant arrays of a characters with different lengths, declared in the *task function* which evaluated the behaviour of messages of different sizes (ID 0 and 1). Another measurement was taken with a structure stored at the memory returned by *get_buffer_ptr(int msgID)* and filled with data during the execution (ID5). Each measurement was performed using a different msgID and each used a different number of buffers which evaluates the assumptions, that neither the ID of a message nor the round it is stored in do not make a difference for the execution time.

Results for ID0 and ID1 showed an execution time of 2081 cycles in each of the 4 times the functions were executed (once per round). ID5, which passed a pointer to a structure instead of an array, showed an execution time of 2080 cycles. This 1 cycle difference originated from different code being generated by the compiler depending on whether to pass the pointer directly or the array. The different instructions generated can be seen in Table 7.2. Measuring these 2 function calls with optimizations (O2) showed the same execution time of 896 cycles. In both cases the memory addresses were kept in a local register. When no optimizations were enabled they were not kept in a register but always fetched from memory even for subsequent accesses.

With a local variable	With memory access
ld [%fp + -28], %g1	sethi %hi(0x8f001000), %g1
st %g1, [%fp + -40]	or %g1, 0x3f0, %g1
	ld [%g1], %g2
	ld [%fp + 0x44], %g1
	add %g1, 4, %g1
	sll %g1, 2, %g1
	add %g2, %g1, %g1
	ld [%g1 + 4], %g1
	ld [%g1 + 8], %g1
	st %g1, [%fp + -32]

Table 7.1: Memory access optimization

A trivial optimization is the elimination of all multi level pointer indirections by storing the last of these pointer accesses in a local variable if it is dereferenced more often than once. Using the general purpose registers removes the necessity for this and is therefore not required when the compiler is allowed to perform optimizations. Storing the pointers in a variable subsequently removes many memory accesses. An example for the difference between the code pieces `tmpPtr=oldPtr` and `tmpPtr=pxCurrentTCB->msgBufferArray[ID]->BufferArrayPtr` is displayed in Table 7.1. As it can be seen a second access to this pointer would benefit dramatically from this small optimization.

The execution time of this function depends on the *MAX_IPC_MSG_SIZE* constant defined in the config.h header. Changing this will result in a different loop bound used for copying content, leading to a different execution time of the function. Measurements with different *MAX_IPC_MSG_SIZE* values showed that the execution time can be expressed by the formula $97 + MAX\_IPC\_MSG\_SIZE * 31$.

task1+164 add %fp, -56, %g1	task1+520 mov 5, %o0 ! 0x5
task1+168 mov 1, %o0	task1+524 ld [%fp + -4], %o1
task1+172 mov %g1, %o1	task1+528 mov 0x11, %o2
task1+176 mov 0x11, %o2	task1+532 call 0x8e001310 store_msg
task1+180 call 0x8e001310 store_msg	task1+536 nop
task1+184 nop	

Table 7.2: Different execution traces of store_msg(0) vs store_msg(5)

7.6.2 Results of the send_msg() Function

send_msg(int msgID) was measured using 6 different msgIDs. Each execution showed a constant execution time of 267 cycles. To show that using different buffers does not change the execution time, messages with 1 (twice),2,3,4 and 5 buffers were used. The measurement spaned 4 execution rounds, therefore all 3 possible cases of buffer handling were observed. This is sufficient although the wrap-around of the messages with 4 and 5 buffers was not observed. The buffer handling does not in any way depend on how the message is stored or its content making more execution rounds unnecessary. Also the wrap-around was observed for the other message IDs, therefore all possible cases have been observed nevertheless.

These cases were:

- Only 1 buffer is used so do not move to the next.
- More than 1 buffer is used so move to the next.

- More than 1 buffer is used and a wrap-around is performed on account of already having used all buffers in this execution round.

An earlier version of the operating system experienced jitter depending on whether or not it moved to the next buffer or wrapped around. This originated from the different length of *if* and *else* paths. Since no conditional move operation is possible the *if-else* paths can not simply consist of sequential code and a conditional move instructions and therefore have to branch. Tfhe *if* path has to contain a jump to the next instruction after the *else* path. It was possible to convert the conditions to be exclusive transforming the *if*-*else* paths into *if*-*if* paths which removed that difference. This is shown in Table 7.3. The left hand side displays the assembler code of the *if* part of an *if-else* code piece. Two branching instructions are seen one for the false path jumping to the *else* part (task1+96) and one at the end of the *if* part (task1+112) to jump over the *else* part. Since the *else* path does not need to skip the next instructions but can continue by simply loading the next instruction it does not need the branch and *nop* instructions. On the right hand side the *if* part in the case of an *if-if* code piece is seen. Only the branching instruction needed to skip the *if* part is left. The former *else* part is evaluated by another *if* and therefore is not skipped. The need to evaluate the expression twice increases the execution time, but no branch instruction is needed. If both blocks have the same execution time the execution time difference of the choices is eliminated.

Another measurement was performed with -O2 optimization levels to see how this coding style is affected by optimizations. Tests revealed a jitter between 105 and 112 cycles, depending on how many buffers were used for each message and how the buffers were handled:

- Moving to the next buffer resulted in a 105 cycles execution time.
- Moving to the same (only) buffer again took 108 cycles.
- Setting to the first buffer again when more than 1 buffer was used resulted in an execution time of 111-112 cycles.

This jitter is similar, albeit a bit larger, to the jitter observed before using the *if-if* paths.

ldub [%fp + -15], %g1	ldub [%fp + -15], %g1	
sll %g1, 0x18, %g1	sll %g1, 0x18, %g1	
sra %g1, 0x18, %g1	sra %g1, 0x18, %g1	
cmp %g1, 0	cmp %g1, 00	
be 0x8e001a60 task1+120	be 0x8e001a60 task1+120	*branch if false*
nop	nop	
ldub [%fp + -13], %g1	ldub [%fp + -14], %g1	
stb %g1, [%fp + -15]	stb %g1, [%fp + -15]	
b 0x8e001a68 task1+128		*skip the else code*
nop		
position task1+120:	*position task1+120:*	
start of else part	*start of next if*	
position task1+128:		
first instruction after else part		

Table 7.3: *if* code part of *if then-else* vs. an *if-if* code piece

7.6.3 Results of the get_buffer_ptr() Function

This *get_buffer_ptr(int msgID)* function was measured with msgID 2,3 and 4. Each execution took exactly 61 cycles and was independent of the message ID and the round it was executed in.

7.6.4 Results of the get_msg() Function

This function *get_msg(int msgID)* was executed 6 times per execution round (msgID 0 to 5). Each execution took exactly 71 cycles and was independent of the message ID and the round it was executed in.

7.6.5 Results of the ipc_send_msg() Task

Since *ipc_send_msg* is a task and not a function, that could be measured from the point it is called to the instruction after it returned, the execution was measured from entering the infinite loop to the execution of the *task_finish()* call and to the time the idle task is started again. This measurement included the time used by the context switch to the idle task.

This measurement unfortunately experienced a 5 cycle jitter. This occurs because a prior execution could have been interrupted before any of 4 instruction, but the execution is always interrupted before the execution of the $i++$ instruction. This unfortunately introduces a 5 cycle measurement uncertainty. The assembler code of the loop is shown in Table 7.4.

The execution until the break point in the infinite loop took between 765 and 770 cycles. Measuring the execution time from before first instruction of the task until the instruction before the call to *task_finish()* showed an execution time of exactly 61.

Because the measurements of the *task_finish()* call showed the same 5 cycles jitter it is concluded that the jitter originates from the measurement uncertainty and not from a difference of the *ipc_send_msg* to the other tasks.

Instruction	Duration in cycles
idletask+4 ld [%fp + -4], %g1	2 cycles
$<$ *breakpoint* $>$idletask+8 inc %g1	1 cycle
idletask+12 st %g1, [%fp + -4]	3 cycles
idletask+16 b 0x8e0023b8 <idletask+4>	1 cycle
idletask+20 nop	1 cycle

Table 7.4: Assembler code of the infinite loop

7.7 I/O Message Base Accesses

The functions measured and their parameters were:

- *write_io_message(int IOMsgID, void *sourcePtr, int data_size_in_bytes)*
- *read_io_message(int IOMsgID, void *sinkPtr, int *dataSizePtr)*

The access to the I/O message base was measured for two IDs which store messages of different sizes (constant char arrays allocated on the stack). These functions use bounded loops and do not take the structure, nor content, of the messages into account. This way the execution time should not vary with the size of the message. Each function was observed over four execution rounds. This is sufficiently long because the system state relevant to these functions does not change between execution rounds.

Each execution of *write_io_message()* took exactly 1761 cycles. Exactly the same was the case for the *read_io_message()* function. Although this looks like a remarkable coincidence it is actually a result of the completely symmetric nature of these 2 functions. The execution time of these functions depends on the *MAX_IO_MSG_SIZE* constant defined in the config.h header. Changing this results in a different execution time because of a bounded loop used for copying content. Again, as with the *store_msg()* function, the different handling of pointers and char arrays could be observed in a different experiment. When a pointer was used execution time decreased to 1760 cycles. Measurements using different *MAX_IO_MSG_SIZE* values revealed that the execution time depending on this constant can, for both functions, be calculated with the formula $97 + MAX\_IO\_MSG\_SIZE * 26$.

7.8 Summary

The measurements have evaluated the temporal properties of the operating system functions. Every function was tested using multiple test data to explore the impact of function parameters and system state on the execution times. These results show that the execution time of most functions is indeed constant and that the functions do not depend on runtime properties. All functions that depend on constants have been measured with varying values to determine a formula that can be used to calculate the execution time depending on that constant beforehand.

Insights into cases, where special attention for writing C code with good temporal characteristics is required, have been gained, as with the *if-else paths* and the pointer dereferences optimization.

Unfortunately some parts of the system could not be measured with 1-cycle accuracy, but the results of these parts were always within the accuracy of measurements. It is therefore concluded that the goal of providing functions with constant execution times has been reached. Knowing the invocation times of the tasks due to the time-triggered scheduler and providing operating system functions that always have, a-priori known, constant execution times makes it easy to argue about the temporal characteristics of a system using this real-time operating system.

The measurement results for all analysed functions and the different configurations that were used are summarized in Table 7.5.

Measurement Results		
Function	Configuration	Cycles
context switch	*IPC_SCHEDULE*	1029
	!*IPC_SCHEDULE*	990
	-“- & dummy code	1029
	-“- & O2	$770 - 773$
task_finish	*TTSCHEDULER*	$768 - 773$
	COOPERATIVE	781
ipc_send_msg		61
get_msg		71
get_buffer_ptr		61
store_msg		$97 + MAX\_MSG\_SIZE * 31$
send_msg		267
	O2	$105 - 112$
write_io_message		$97 + MAX\_IO\_MSG\_SIZE * 26$
read_io_message		$97 + MAX\_IO\_MSG\_SIZE * 26$

Table 7.5: Summary of measurement results

CHAPTER 8

Conclusion

This thesis has demonstrated the development of a RTOS whose temporal characteristics are fully known before run-time. The time-triggered approaches for the scheduler, I/O as well as using a shared memory IPC-approach combined with the circular buffer guarantee tight temporal control over all aspects of the operating system. Also no conditional statement in the operating system code depends on run-time properties and all operating system functions have constant execution times.

The low number of branches allowed the application of white-box testing based measurement to determine the WCET of the operating system functions.

The measurements of all operating system functions during the real-time phase demonstrated that the execution times of all functions are within the error of measurement. This demonstrates that the goal of writing a time-predictable operating system has been reached. This was achieved by avoiding algorithms and programming styles known to be detrimental to worst case behaviour and instead using WCET-oriented programming patterns, as well as borrowing some ideas from the single path approach like to sequentialize execution of branching code.

Interesting insights into traps and pitfalls for writing time-predictable applications, like the *if-if* paths and the pointer optimization, have been gained. The operating system is easily modifiable and provides different configurations that can be useful in different deployment scenarios. Also the WCET analysis of application code does not need to analyse the operating system code because of their complete temporal characterization by the constants and formulas determined by the measurements.

8.1 Future Work

Porting the operating system to a CPU architecture that provides predicated instructions to test the portability of the operating system would be an interesting experience. Also using the same operating system code on different platforms provides an excellent opportunity for comparing the different effort required to develop applications with constant execution times.

An interesting extension would be to encapsulate the whole task state in its TCB. This would make it easier to track the system state and allow synchronization between nodes and/or recovery mechanisms easy by exchanging the TCB content.

It could be beneficial to be able to execute non real-time tasks whenever no real-time task is required to run. This could be done by invoking a non real-time scheduler whenever a real-time task reaches the end of its execution. The non real-time tasks could use the time available until the next real-time task is executed to increase the CPU utilization.

As a result of the combination of the constant execution times and the extensible LEON3 CPU core, testing the impact of CPU features on the execution time can be done easily and would constitute an instructive experiment.

Bibliography

[Bar09] Richard Barry. *Using the FreeRTOS Real Time Kernel - a Practical Guide*. 1.0.5 edition, 2009.

[Cor03] On-Line Applications Research Corporation. *RTEMS C User's Guide*. 4.6.5 edition, 2003.

[CP01] Antoine Colin and Isabelle Puaut. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. In *ECRTS*, 2001.

[DHPS03] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor Support for Temporal Predictability -- The SPEAR Design Example; The SPEAR Design Example. *Real-Time Systems, Euromicro Conference on*, 2003.

[Dij02] Edsger W. Dijkstra. *Cooperating sequential processes*. 2002.

[Gai10] Aeroflex Gaisler. *GRLIB IP Core User's Manual*. 2010.

[HS02] Pillai Padmanabhan Huang, Hai and Kang G. Shin. Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, 2002.

[Inc92] SPARC International Inc. *The SPARC Architecture manual Version 8*. 1992.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[KPD08] Guenter Khyo, Peter Puschner, and Martin Delvai. An Operating System for a Time-Predictable Computing Node. In *Software Technologies for Embedded and Ubiquitous Systems, 6th IFIP WG 10.2 International Workshop (LNCS 5287)*, 2008.

[KR93] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In *Real-Time Systems Symposium, 1993., Proceedings.*, 1993.

[LGZ+09] Mingsong Lv, Nan Guan, Yi Zhang, Qingxu Deng, Ge Yu, and Jianming Zhang. A Survey of WCET Analysis of Real-Time Operating Systems. In *Proceedings of the 2009 International Conference on Embedded Software and Systems*, 2009.

[MM92] T.J. Marlowe and S.P. Masticola. Safe optimization for hard real-time programming. In *Systems Integration, 1992. ICSI '92., Proceedings of the Second International Conference on*, 1992.

[PB00] Peter Puschner and Alan Burns. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. volume 18, 2000.

[PB02] Peter Puschner and Alan Burns. Writing Temporally Predictable Code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[PK10] Peter Puschner and Raimund Kirner. Zeitanalyse sicherheitskritischer Echtzeitsysteme, 2010.

[Pro08] Daniel Prokesch. Patterns of WCET-Oriented Programming, 2008.

[PS01] Frederick M. Proctor and William P. Shackleford. Real-time Operating System Timing Jitter and its Impact on Motor Control. In *Proceedings of the SPIE Sensors and Controls for Intelligent Manufacturing II*, 2001.

[Pus99] Peter Puschner. Real-time performance of sorting algorithms. *Journal of Real-Time Systems*, 16(1), 1999.

[Pus03] Peter Puschner. Algorithms for dependable hard real-time systems. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, 2003.

[Pus05] Peter Puschner. Experiments with WCET-Oriented Programming and the Single-Path Architecture. In *WORDS*, 2005.

[Rei93] J Reisinger. Konzeption und Analyse eines zeitgesteuerten Betriebssytems für Echtzeitanwendungen,Ph.D thesis, Technisch-Naturwissenschaftliche Fakultät,Technische Universität Wien, 1993.

[RS94] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of the IEEE*, 1994.

[SEGL04] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Static Timing Analysis of Real-Time Operating System Code. In *ISoLA*, 2004.

[SR04] John A. Stankovic and R. Rajkumar. Real-Time Operating Systems. *Real-Time Syst.*, 28(2-3), 2004.

[Sta01] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 4th edition, 2001.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7, 2008.

[ZKW$^+$05] Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, and Frank Mueller. Improving WCET by Optimizing Worst-Case Paths. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, 2005.

APPENDIX A

Error Codes

The following table shows the meanings of the error codes categorized by which module generates them.

Error Code	Meaning
	Scheduler
1	the wait_sync_msg task returned
2	taskID negative
6	pointer to the tasks function is NULL
24	timer value is negative
28	sync msg not received
43	current TCB Pointer is NULL in ISR
	Task
3	taskID < 0
4	type of the task < 0
5	stackSize $<= 0$
6	pointer to the tasks function is NULL
7	pointer to the tasks init function is NULL
8	allocate_TCB returned a NULL pointer
9	allocate_stack returned a NULL pointer
10	parse schedule returned 0, taskID not found in scheduling array
11	TCBPtr is NULL
	IPC

12	msgID negative or larger than MAX_MSG
13	number of buffers negative
14	malloc returned a NULL pointer
15	Buffer not large enough to store msg
16	tmpBufferPtr is still NULL
40	trying to set the next IPC msgID to < 0 or larger than MAX_MSG
41	msgID is bigger than MAX_MSG in get_buffer_ptr
42	msgID is negative in get_buffer_ptr
Temporal Firewall	
17	IOMsgID bigger or equal the max number of msges or negative
18	trying to read a msg with zero or negative size
25	trying to write a msg with zero or negative size
36	size in the IOMsgbase is now negative although dataSize should be ok
37	iosize value in the msg base is negative or 0
TTIO	
19	trying to set the next msgID to < 0 or larger than MAX_IO_MSG
26	next_msg_to_process invalid value
35	size of the msg we want to read is 0 or negativ
39	trying to write a msg which is larger than MAX_IO_SIZE
Memory	
23	Not enough memory available, exiting now
Miscellaneous	
27	MAIN returns
7−Segment Display	
29	can not display such a hex value on a single 7segment, value bigger than 0xF
30	this 7seg display does not exist
UART	
31	UART Overflow flag set
33	UART Parity error flag set
34	UART Frame error flag set
38	There seems to be data in the FIFO although we are initializing
GPIO	
32	irq 1 is reserved for the gpio0 callback -pin gpio1 can not have an IRQ
Leds	
21	that many leds do not exist

22	led does not exist
Button	
20	button does not exist

APPENDIX B

Measurement Application Code

```
 static portINT task1_init_function(tskTCB* TCBPtr) {
        TCBPtr->msgBufferArray[0] = msg_create(0, 2,
        (unsigned portINT) (64* sizeof(portINT)));
        TCBPtr->msgBufferArray[1] = msg_create(1, 4,
        (unsigned portINT) (64* sizeof(portINT)));
        TCBPtr->msgBufferArray[2] = msg_create(2, 3,
        (unsigned portINT) (64* sizeof(portINT)));
        TCBPtr->msgBufferArray[3] = msg_create(3, 1,
        (unsigned portINT) (64* sizeof(portINT)));
        TCBPtr->msgBufferArray[4] = msg_create(4, 5,
        (unsigned portINT) (64* sizeof(portINT)));
        TCBPtr->msgBufferArray[5] = msg_create(5, 1,
        (unsigned portINT) (64* sizeof(portINT)));
        return 1;
}

static portINT task2_init_function(/*@unused@*/tskTCB* TCBPtr) {
        return 1;
}

static portINT task3_init_function(/*@unused@*/tskTCB* TCBPtr) {
        return 1;
```

```
}

static void task1(void) {

        int i;
        char toggle=1;
        portCHAR msg0[] = "msgID1\\0";
        portCHAR msg1[] = "msgID1longstring\\0";
        portCHAR *msg2;
        portCHAR *msg3;
        Test *msg4;

        while (TRUE) {
                task_finish();
                store_msg(0,msg0,7  );
                send_msg(0);
                store_msg(1, msg1, 17);
                send_msg(1);
                msg2 = get_buffer_ptr(2);

                for (i=0; i< 6;i++) {
                        *(msg2+i) = msg0[i];
                }

                send_msg(2);
                msg3 = get_buffer_ptr(3);

                for (i=0; i< 17;i++) {
                        *(msg3+i) = msg1[i];
                }

                send_msg(3);
                msg4 = (Test*)get_buffer_ptr(4);

                for (i=0; i< 6;i++) {
                        msg4->teststr[i] = msg1[i];
                }

                msg4->value1 = 10;
                msg4->value2 = 20;
                send_msg(4);
                store_msg(5,msg4,17);
                send_msg(5);
        }
        return;
}

static void task2(void) {

        portCHAR *msg1;
        portCHAR *msg2;
        portCHAR *msg3;
        Test *msg4;
        Test *msg5;
        int size1, size2, size3,size4,size5;
        portCHAR iomsg0[] = "IOmsgID1\0";
        portCHAR iomsg1[] = "IOmsgID1longstring\0";

        while (1) {
                task_finish();
                msg1 = (char*) get_msg(0,&size1);
                msg2 = (char*) get_msg(1,&size2);
                msg3 = (char*) get_msg(2,&size3);
                msg4 = (Test*) get_msg(4,&size4);
                msg5 = (Test*) get_msg(5,&size5);
                write_io_message(1, iomsg0, 9);
                write_io_message(2, iomsg1, 19);
        }
        return;
}

static void task3(void) {
        portINT iosize1, iosize2;
        portCHAR iosink1[MAX_IO_MSG_SIZE];
        portCHAR iosink2[MAX_IO_MSG_SIZE];

        while (TRUE) {
                task_finish();
                read_io_message(1, iosink1, &iosize1);
                read_io_message(2, iosink2, &iosize2);
        }
        return;
}
```