INSTITUTE FOR REAL-TIME COMPUTER SYSTEMS
TECHNISCHE UNIVERSITÄT MÜNCHEN
PROFESSOR SAMARJIT CHAKRABORTY

Das ist ein etwas längerer Titel dieses leeren Diplomarbeitsgerüsts

Nadja Peters

Master's Thesis

Das ist ein etwas längerer Titel dieses leeren Diplomarbeitsgerüsts

Master's Thesis

Supervised by the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr. sc. Samarjit Chakraborty

Executed at Siemens AG

Advisor: Philipp Kindt, Guillaume Pais, Christian Bachmann

Author: Nadja Peters
Arcisstraße 21
80333 München

Submitted in August 2013

Acknowledgements

Vielen Dank . . .

München, im Monat Jahr

Contents

List of Figures

List of Tables

Abbreviations

API Application Programming Interface

BKL Big Kernel Lock

CFS Completely Fair Scheduler

CPU Central Processing Unit

eCOS Embedded Configurable Operating System

EMIO Extended Multiplexed I/O

FIFO First-In-First-Out

FPGA Field Programmable Gate Array

FSBL First Stage Boot Loader

GIC Global Interrupt Controller

GPIO General Purpose I/O

IRQ Interrupt Service Request

ISE Integrated Software Environment

ISR Interrupt Service Routine

I/O Input/Output

JTag Joint Test Action Group

LED Light Emitting Diode

MAC Macintosh

MIO Multiplexed I/O

MMU Memory Management Unit

MPU Memory Protection Unit

MTD Memory Technology Device

OS Operating System

POSIX Portable Operating System Interface

PID Process Identifier

PL Programmable Logic

PS Processing System

QoS Quality of Service

QSPI Quad SPI

RAM Random Access Memory

RCS Real-Time Computer Systems

ROM Read Only Memory

RR Round Robin

RTAI Real-Time Application Interface

RTOS Real-Time Operating System

SD Secure Digital

SDK Software Development Kit

SMP Symmetric Multiprocessor

SPI Serial Peripheral Interface

SSBL Second Stage Boot Loader

UART Universal Asynchronous Receiver Transmitter

UBI Unsorted Block Images

UBIFS UBI File System

USB Universal Serial Bus

Abstract

Die Kurzfassung ...

1 Introduction

Nowadays, electrical systems have grown so complex that they are usually managed by OSes. An operating system is a software program which provides access to the underlying hardware. Its purpose is to manage hardware resources - for example Central Processing Unit (CPU) time, memory or Input/Output (I/O) access - as well as system and user processes efficiently. Depending on the application there are different kinds of OSes. In *General Purpose OSes* like Windows, Macintosh (MAC) OS or Linux, the main goal is to maintain fairness between different users or processes. Consequently, every user or process should get an equal time slice of the available CPU time or other shared resources like Random Access Memory (RAM). OSes are also used in embedded systems. In contrast to General Purpose OSes, operation systems for embedded devices often have to run under special conditions, e.g. limited memory size or low power consumption.

1.1 Real-Time Systems

A special kind of embedded systems are Real-Time Operating Systems (RTOSes) which are designed to meet specific deadlines. The main property of an RTOS is determinism (and not necessarily high-speed performance). Real-time systems are divided into three classes, depending of the consequence caused by missing a deadline:

- Hard
- Firm
- Soft

In hard real-time systems, missing a deadline causes system failure and is not tolerable. An example is the engine control system of a car which can be damaged or cause an accident because of delayed signals. Another example is the release of an airbag in a car. It has to be triggered immediately when a crash happens, any delay could cause the loss of lives. More applications can be found in medical systems or industry processing control. In firm real-time systems, deadline misses may occur at rare intervals and cause loss of Quality of Service (QoS), but not a complete system failure. In soft real-time systems, deadlines can be missed but decrease the QoS. These kind of systems are usually used for application with a continuous data flow like multimedia streaming applications or on-line reservation systems.

The class of a system always depends on the application. To meet the given requirements, the underlying hardware and - often - an OS have to be chosen appropriately. In embedded systems, Linux is widely used because it is free of charge, has a large community and therefore a high level of support. Moreover, it is comfortable to use as it embeds features like a command terminal, flexible module integration and a large number of available hardware drivers. Yet, resulting from its complexity, the downside of Linux is still the lack of real-time capability. Although extensions like *Real-Time Application Interface (RTAI)* [14], *Xenomai* [20], the *PREEMPT_RT* patch and many commercial Linux distribution exist, reliability of the system is hard to proof. To achieve hard real-time performance, usually special OSes are used. Those can be based on a real-time kernel design or light-weight OSes like for example *µCOS* [10] or *FreeRTOS* [16]. They have very small memory footprint, fast boot time and no background services (deamons) which could unexpectedly disturb the execution of real-time tasks. Therefore the execution time of tasks or the interrupt latency are highly predictable. The jitter (variance of latencies) is very low compared to a system like Linux. The disadvantage of these systems is that every change in the underlying hardware requires a reconfiguration of the OS. Furthermore, a high-level communication with the system is not provided and has to be implemented if needed. Consequently, an OS which can provide deterministic timing as well as the convenience of a more advanced system is desirable.

To make the right choice regarding an OS, it is necessary to know how exactly the given systems differ in performance. Obviously, a Linux-based system will have higher latencies and jitter. Still, recently especially the Linux PREEMT_RT patch (further: *LinuxRT*) was improved a lot in the matter of predictability and is easy to apply to an existing Linux distribution. Whenever possible, it would be chosen over a light-weight system by a developer. To make this possible, some guidelines in the actual performance of the OSes are desirable. How does LinuxRT perform compared to a light-weight OS? When can LinuxRT be used and when is it inevitable to use a light-weight OS? To answer these questions, LinuxRT has to be compared to a suitable light-weight OS. In the past decade, FreeRTOS has grown to be a popular RTOS solution. It is supported on many platforms, is freely available and already used in different market sectors, for example toys, aircraft navigation or engine control. Therefore, it is a good candidate to be used as reference. The next point to consider is how OSes can be compared to each other.

1.2 Benchmarking of RTOS

There are quite many criteria which can be applied to benchmark OSes. For RTOS, obviously, it is most important to consider features typically used in real-time applications. Those are mainly task synchronization features, e.g. semaphores, message queues or signals. Moreover, interrupt latency is crucial because interrupts usually wake up important tasks in RTOS. Another typical application is the periodic invocation of tasks. As these tasks have to be scheduled as precisely as possible, the jitter is an interesting metric. More key features are preemption time of tasks and deadlock breaking time. These two metrics

indicate the capability of the system to interrupt a low priority task by a task with higher priority. Boot time can also be important in real-time systems as it is not acceptable to wait 15 seconds for the engine to run after starting the car. Further, some systems support memory access supervision by a Memory Management Unit (MMU). As RTOS are usually used on embedded devices, the memory footprint is also of large interest.

One challenge is to choose the right criteria from the ones mentioned above. Another one is to measure values which are as accurate and comparable as possible. Therefore, a way of time measurement has to be found, which can be applied on both systems with the least possible interference on the system.

1.3 Related work

Performance evaluation has already been done on different platforms and with different OSes. The first attempt to developing a real-time benchmark was by Kar and Porter in 1989 by introducing the *Rhealstone Benchmark* [6] [15]. This benchmark is based on six parameters (for details refer to **??**) the resulting value is calculated from. The Rhealstone Performance Number is a weighted mean of all parameters. They implemented the benchmark under iRMX.

Another and more recent comparison of operating systems was performed for a set of OS suitable for small microcontrollers in 2009 [18]. In this paper, several RTOS are introduced, but only four of them are chosen to be investigated in detail. The choice is based on available support, documentation, scheduling type and more. For the RTOSes, algorithms on how to compare task switching, message passing, semaphore passing and memory allocation are presented. In [3], memory footprint and context switching is compared between μCOS and XilKernel[1)] on the softcore processor Microblaze. Three different OSes - Xenomai, LinuxRT and eCos [1] - are compared in [9]. The performance metrics are similar to the Rhealstone benchmark. Depending on the application, eCos and LinuxRT perform better than Xenomai. The authors of [8] and [19] evaluate the real-time performance of LinuxRT compared to standard Linux. In both works, a sample program is run solely, with CPU and I/O load. Whereas in [8] the focus is mainly on the effect of real-time priorities, in [19] different period lengths and multi-processor effects are also investigated. The results show a significant reduction in jitter when using LinuxRT.

[Related work on OS jitter]

1.4 Contribution

In this work, a guideline on the choice of an operation system for specific use cases is presented. Therefore, a set of benchmarks is defined based on the Rhealstone benchmark

[1)] XilKernel [21] is an RTOS developed by Xilinx.

proposed by Kar and Porter [6] [15]. These benchmark metrics are extended by the boot time and task periodicity jitter which are crucial for many applications. A detailed instruction on how to implement the benchmark is provided. Further, a model is introduced on how to calculate OS latency from the obtained benchmark metrics. A special focus is put on the operation systems LinuxRT and FreeRTOS. FreeRTOS is the most important light-weight RTOS nowadays. As there is a large interest in using more comfortable OS like Linux, the goal is to clearly define the limits of LinuxRT. Moreover, the sources of OS jitter shall be detected and reduced as far as possible. The optimum result is a LinuxRT configuration with comparable benchmark results as FreeRTOS.

The remainder of the work is organized as follows. In chapter 2 a detailed background on the topic is provided. Chapter 3 describes the performed measurements in detail. Further, in Chapter 4 the results of the measurements are presented. In the last Chapter 5 the contribution of this work is summarized.

2 Background on Operating Systems

This chapter provides an deeper insight into the background of the topic. It gives details on RTOSes in general, on scheduling in FreeRTOS and LinuxRT, and the RT patch. Moreover, the causes of OS jitter are described, especially the timer interrupt. Finally, the main design features of real-time programs are discussed and the benchmarking metrics for the two OS are introduced.

OSes manage the hardware resources - CPU time, I/O and memory access - for different tasks and process. Dependent on its level of functionalities, OSes have different levels of complexity. While a system like FreeRTOS has a limited number of features, Linux offers a variety of services which run in the background.

2.1 The Scheduler

Every task or process created by the kernel or a user, is managed by the operation system. The scheduler manages the tasks and decides whether a task switch is performed. The decision is based on the priority and the current state of the task.

2.1.1 Task States

Dependent on events or resource availability, tasks can enter different states. In the following, the state flow used in FreeRTOS [16] is described. The state flow in Linux is related, but extended by other states which are not of interest for the further work.

[Task state picture]

Ready When a task is schedulable, it is in the state *Ready*. A process enters this state when it is first created, has been unblocked by an Interrupt Service Routine (ISR) or an other task or when resources it was waiting for become available.

Running Tasks in this state get access to the CPU, they are currently being executed. This state can only be entered from tasks in the Ready state.

Blocked Tasks enter the *Blocked* state when they are waiting on a queue or a semaphore or another event. Tasks can switch from this state to the Ready state when the according event occurs, e.g. another tasks releases the semaphore.

Suspended A task can be suspended by itself or by another task. This task cannot wake up on an event but has to be explicitly unsuspended to reenter the Ready state.

2.1.2 Timer Tick

The timer tick invokes the scheduler and can be programmed in FreeRTOS as well as in Linux. The tick is triggered by a timer interrupt a definite number of times per second - usually 100 times. If the tick period is too low, tasks may finish running a long time before the next tick occurs. Therefore, processor time might be wasted. On the other hand, if the tick period is too high, the OS overhead on the system is too large. The scheduling routine and an internal counter update are invoked in FreeRTOS in the timer ISR. Further, the timer tick causes latency in the system performance called *OS jitter*.

[Picture for timer tick]

2.1.3 Scheduling Policies

The scheduler decides which task is allowed to enter the Ready state. The decision is based on the scheduling policy of the OS. There are mainly two different types of scheduling: *Preemptive* and *cooperative* scheduling. In cooperative scheduling, tasks cannot be interrupted, they have to release the processor voluntarily. The scheduling policy is crucial for RTOSes because it determines the handling of real-time tasks. Obviously, real-time tasks usually have the highest priority and should not be preempted by tasks with lower priority.

Idle Task

The *Idle Task* is scheduled when no other tasks are present in the Ready queue because the processor is not allowed to run out of work. It usually has the lowest possible priority. This task is often used to put the CPU in a low-power mode (scaling down frequency or executing the *halt* instruction). Moreover, it can be used to perform background processes or as indicator for spare time

Scheduling in FreeRTOS

Scheduling in FreeRTOS [16] can be either cooperative or preemptive and is purely priority based. The maximum priority can be defined by the user. The memory footprint grows

with increasing priority number, so it is recommended to choose only as many priority levels as needed. The task with the highest priority which is in the Ready state will be scheduled. Tasks can have the same priority if desired. Priorities can be changed in runtime.

All tasks are managed in doubly linked lists dependent on their state. The Ready state is implemented as a list array indexed by the priority level. When the scheduler is invoked, it first updated the Ready lists and then schedules the next task. In case there are multiple tasks in one list, the Round Robin (RR) algorithm is used to pick the one.

[Picture Ready Queue]

Scheduling in Linux

Scheduling in Linux [7] [5] is more complex than in FreeRTOS. Three scheduling classes are used to determine which task will be selected: *sched_rt*, *sched_fair* and *sched_idletask*. Sched_rt implements the scheduling of real-time tasks and has the highest priority. General purpose tasks are using the sched_fair class and the remaining class is used by the idle task.

sched_rt Linux real-time tasks are always scheduled prior to any other task. The default priority levels range from 0 to 99 where 99 is the highest possible value. The maximum priority can be configured by the user. The run queue is implemented comparable to the one of FreeRTOS. In Linux, there are two different scheduling policies for real-time tasks: RR (preemptive) and First-In-First-Out (FIFO) (cooperative).

sched_fair For the sake of completeness, the fair scheduling algorithm is briefly described in this section. Linux was originally developed as a General Purpose OS, so its scheduling algorithm is optimized to treat all tasks as fair as possible. The current scheduler is called *Completely Fair Scheduler (CFS)* and was introduced in Linux kernel 2.6.23. It is implemented as a Red-Black tree, which is self-balancing (no path is more than twice as long as any other). An element of the tree can be accessed in in O(log n), where n is the total number of nodes in the tree.

The prioritization of the CFS is based on the time the tasks have already been executed. The lesser the time compared to the other task, the higher the chance to get CPU access. The most-left node is scheduled on a context switch. Further, the priority of an element can be influenced using the *nice* command. It puts a weight on the according node which will change the priority relative to the other nice values.

2.2 Intertask Communication and Synchronization

In OSes usually not only one but many different tasks are active. These tasks need ways to communicate with each other. Moreover, race conditions on critical resources, e.g. concurrent memory access, needs to be prohibited. The mechanisms introduced in the following are available for FreeRTOS as well as for Linux.

2.2.1 Events

In many cases tasks need to wait until another task completes execution. Instead of polling a resource (actively waiting and therefore wasting CPU resources), tasks usually wait for an event to happen. Therefore, a task blocks or sleeps on a specific signal. When another task completes execution, it triggers the signal and wakes the task waiting on it. Such a signal can also be triggered by an ISR

2.2.2 Memory access

Concurrent memory access of multiple tasks can have undesired results as illustrated in the following example:
Person A and person B want to withdraw money (both 500 Euros) from a bank account via an ATM at the same time. Task A starts processing the request of person A. It reads the currently saved value X from memory and subtracts 500 Euros. At this point it may run out of time and task B is scheduled. Task B also reads the currently stored value X from memory (which has not yet been updated) and subtracts 500 Euros. Like A before, task B is interrupted at this point and task A is scheduled. A writes back the new value X and finishes execution. Then, B also writes back its value X. Obviously, the value of X - 500 Euros is wrong, it should be X - 1000 Euros.

To prevent such inconsistencies, the memory has to be locked by the task until the correct value has been written back. So called *mutexes* can be used for this purpose. When a mutex (short for mutual exclusion) is taken by a task, every other task which tries to access this mutex will be blocked. By releasing the mutex, the waiting tasks are unblocked. Mutexes are a special form of semaphores. Semaphores allow a definite number of tasks to access a particular resource, for a mutex it is only one.

2.2.3 Message Passing

Besides controlling the execution flow of other tasks using the mentioned features, tasks can also pass messages between each other to exchange information. Usually, one task blocks on a message queue until another task puts a message in this queue. That event wakes up the blocking task and it retrieves the message. This mechanism is a mix of signaling event and synchronizing shared memory access.

2.3 Interrupt Handling

Interrupts signals provide an important method for the processor to communicate with peripheral devices. The devices range from mice and keyboards to Ethernet or harddrive controllers. The interrupt signal from the hardware device is connected to the interrupt controller of the processor. This controller signals the processor that an Interrupt Service Request (IRQ) has arrived. Now, as the name suggests, the currently executing process is interrupted, the context saved and the corresponding ISR is loaded to be executed. Because they disturb the execution flow of other processes, ISRs should be kept as short as possible. Typically, only urgent operations are performed in the ISR. For less critical work, e.g. copying data into a transmit buffer of a network device, an extra task is started which can be scheduled later. The minimal version of an ISR should at least contain the acknowledge of the interrupt so the underlying hardware can continue its work. The time between an interrupt being triggered and the first instruction of the corresponding ISR is called interrupt latency.

Interrupts can occur any time and are serviced immediately. Consequently, any code currently running will be interrupted. Some sections in the kernel code have to be run atomically to prevent critical errors in the systems. Such sections are called critical sections. Interrupts are usually disabled on entering a critical section. Therefore, servicing the interrupt is postponed what causes a higher interrupt latency. Interrupts can be prioritized what causes high priority interrupts to disturb the execution of lower prioritized interrupts. This is called interrupt nesting.

2.3.1 Interrupts in FreeRTOS

FreeRTOS itself contains only one interrupt: The timer tick (s. 2.1.2. Other devices and interrupts may be installed on demand, but they are not handled by the OS. It is only in charge of task managing and inter-task communication. For the application development, it means that an interrupt handler has to be written and registered in the Global Interrupt Controller (GIC). As all interrupt handling is up to the programmer, it is easy to keep track of the number of interrupts in the system.

2.3.2 Interrupts in Linux

The interrupts in Linux are integrated into the operation system. Each hardware device needs a driver to communicate with the OS. This driver provides *open()*, *read()* or *write()* functions to access the device. It has to be registered before the device can be used. Drivers can be started at boot time or dynamically be loaded as modules during runtime. The interrupt handler is part of the driver and has to be registered in the kernel as well by calling the function *irq_request()* and on deregistration freed by calling *irq_free()*. When an interrupt occurs, the *do_IRQ()* kernel function occurs which takes care of all ISRs.

[Picture Linux Kernel development, p. 123]

2.4 Delays

Different delays caused by the underlying hardware or the OS are crucial in real-time applications. There are mainly three kind of delays:

1. Delays from interrupts
2. Delays from task execution and synchronization
3. Delays from cache misses

Moreover, the boot time can be crucial in special real-time application. [note on uniprocessor systems]

2.4.1 Delays from Interrupts

The first kind of delays (for details refer to 2.3) can only be prevented by introducing critical sections. However, this may cause higher interrupt latencies and is counterproductive in applications which rely on a fast interrupt response time. Critical sections are mainly found inside of the kernel. As interrupts are caused by I/O devices, the delays from interrupts will increase when the system heavily communicates with a large number of peripherals.

2.4.2 Delays from Cache Misses

The last cause of delays is caching. On one hand, it increases the performance of an application significantly by reducing slow memory accesses. On the other hand, the cache can be used by every process on the system. The pages used by the real-time application could be flushed from the cache which causes hardly predicable caches misses. Yet, software and special drivers can influence the cache behavior if provided by the hardware architecture. Those drivers are available in Linux and can be manually included in FreeRTOS. They allow to lock specific cache pages for critical data or applications, but should be used carefully. Excessive use of page locking eventually has a negative impact on the overall system behavior.

2.4.3 Delays Task Execution and Synchronization

The second source of delays lies in tasks which are currently executing. There are several sources of latencies:
The obvious one is the task switching time when a context switch occurs. This is not

avoidable and happens either when the timer tick occurs (refer to **??**) or when a task blocks, finishes or voluntarily releases the CPU. Another latency related to scheduling is the preemption time. This is the time which it takes the OS to interrupt a low priority task and schedule a high priority task. The interrupt can be caused by giving a semaphore, sending a message or waking up the task from an ISR.

Besides from scheduling, task synchronization causes latencies in a OS. This process causes latencies which are dependent on the implementation and therefore specific for every OS. Examples are semaphores, mutexes, message passing or signals. When a mutex is released, other tasks may be unblocked. This means, that whenever this action happens, the OS has to check whether tasks can be moved from the Blocked state to the Ready state. Moreover, if a task with higher priority was woken up, the current task is preempted and a context switch takes place. Dependent on the application and the right synchronization method, the overall latency in a system can be reduced.

The delays described in this section are CPU bound delays compared to delays caused by acI/O interrupts.

2.4.4 Boot Time

The boot time can be a relevant factor if the system need to start up very fast. When a car is started, the driver does not want to wait 15 seconds until the ignition runs. Obviously, the boot time depends on the memory print of the operation system and on the medium from which the OS is booted. Another important factor is the initialization of hardware structures, e.g Field Programmable Gate Array (FPGA) designs. Moreover, a file system may be needed for the OS kernel to work properly which must be loaded besides the actual kernel. A boot loader is necessary to start an application or an OS. For simple programs or light-weight OS like FreeRTOS usually an First Stage Boot Loader (FSBL) is sufficient. Optionally it can load an FPGA design as well. For more complex systems like Linux [4], the FSBL is used to load a more sophisticated Second Stage Boot Loader (SSBL). Then the SSBL loads the kernel and optionally the initial RAM disk image into memory. After those steps, the kernel can be decompressed and finally boot. While booting, it initializes the file system with the previously loaded image, initializes peripheral devices and finally starts the user space application.

Ways to optimize this process will be discussed later (refer to section 3.2).

2.5 Linux and RT Patch

As Linux is a highly complex operation system, it should be discussed in detail more closely. Especially the differences between standard Linux and LinuxRT [17] will be pointed out. The Linux RT patch aims to make the Linux kernel more preemptive and therefore, significant changes in the kernel structure were made. The most important

ones are removing large critical sections, reducing interrupt latencies and implementing priority inheritance (refer to 2.3 for more details).

2.5.1 Spin Locks

As discussed before, large critical sections reduce the responsibility of a system. On single CPU systems, it is enough to disable interrupts in a critical section, but in multicore systems, concurrent access from multiple CPUs must be prevented as well. Therefore, so called spin locks were implemented. When a spin lock is acquired by one task, another task which tries to take the same spin lock starts spinning in a busy loop. The purpose of this is to protect very short critical sections where a context switch takes more time than waiting for the other task to finish. Yet, spin locks were also used to protect large sections in the kernel and caused big delays. To solve this problem, spin locks were replaced by mutexes when possible which allow the preemption of critical sections.

2.5.2 Threaded Interrupts

One other big factor to reduce latencies was the introduction of threaded interrupts in the RT patch. Originally, interrupt service requests were handled completely in interrupt context. This means that high priority tasks had to wait for low priority interrupts to complete, e.g. disk I/O. A solution to this is moving the work from the interrupt context to an interruptible thread. Therefore, when an interrupt occurs, a working thread in started (or resumed) in the ISR. As default, those threads have a real-time priority (refer to 2.1.3) of 50 in the current implementation. This mechanism allows priority based regulation of the ISR execution because priorities of real-time tasks can be changed dynamically. Moreover, the delay caused by interrupts decreases significantly for high-priority real-time tasks. In case an IRQ has to be serviced immediately, there is still the possibility to set the *IRQ_NODELAY* flag on initialization. As a result, the ISR will not be threaded but proceeded in the original way.

2.5.3 Priority Inheritance

A problem which may occur while synchronizing memory access for tasks with different priorities, is starving a high priority task due to unbounded priority inversion. This can happen in the following situation [17]: Task A to C have different priorities, where A has the lowest and C has the highest. Task A takes mutex M and then is preempted by task B. Task C attempts to access M as well but blocks as it is already taken. Now C is indirectly blocked by B an undefined amount of time, maybe forever.

This situation can be resolved by priority inheritance. With priority inheritance, task A gets a priority boost when C tries to access the semaphore. Therefore, task B is preempted by A and A can leave the critical section. As soon as A releases mutex M, the priority level

is set back to normal. Now C is woken up and can finish its work before B is scheduled again.

2.6 Quantifying an Operation System

3 Measurements

This chapter describes the measurements performed for benchmarking the OSes FreeRTOS and LinuxRT. First, the conditions under which the experiments have taken place are described. This includes hardware platform, time measuring technique and setup of the operation systems under test. After that every measurement is described in detail beginning with the boot time.

3.1 Test Environment and tools

This section contains a description of the test environment - the hardware platform, the development tools and the configuration of the used OSes.

3.1.1 Hardware Platform

The underlying hardware platform is a Xilinx ZC702 Evaluation Board [23] with an Zynq-7000 XC7Z020 [22]. The XC7Z020 chip integrates a Processing System (PS) and a Programmable Logic (PL) on a single die. Two ARM Cortex-A9 MPCore application processors running at 666 MHz are located in the PS. Both cores inherit a 32 KB instruction and a 32 KB data level 1 cache. Moreover, they share a 512 KB level 2 cache and 256 K SRAM. The PL is an Artix-7 from the Xilinx's 7 series FPGA technology and can be used to implement custom hardware designs. Further, the evaluation board provides 128 Mb of Quad SPI (QSPI) flash memory.

The PS uses so called Multiplexed I/O (MIO) pins to connect to peripheral devices. Those pins can be configured to connect either General Purpose I/Os (GPIOs), Ethernet, Serial Peripheral Interface (SPI) and others. The PL and the PS can be connected using the Extended Multiplexed I/O (EMIO) pins. Moreover, the EMIO pins can be used to extend the number of connected peripherals.

When a tool for the Zynq platform is developed, most likely both PS and PL will be included. Therefore, an interrupt generator was implemented in the FPGA design to utilize the PL. It is connected as GPIO device.

The Xilinx Design Suite version 14.4 was used to create the test environment and the tests. The FPGA part was designed and synthesized in Xilinx Integrated Software Environment (ISE). The software was developed using Xilinx Software Development Kit (SDK).

3.1.2 OS Configuration

The operation systems under test are FreeRTOS and LinuxRT. Both need to be configured such that the implemented interrupt generator can be utilized.

FreeRTOS

As already mentioned, interrupt management does not effect the kernel in FreeRTOS. Therefore, the OS itself does not need to be configured. Still, the hardware needs to be initialized appropriately on system startup (refer to 2.3.1). Moreover, FreeRTOS needs an FSBL which can be created by the SDK. The FSBL and the FreeRTOS application must be combined to a boot image. The FreeRTOS version in this project based on version 7.0.2 and is a special port for Xilinx SDK 14.4.

LinuxRT

Xilinx provides their own distributions of embedded Linux which can be compiled for many different platforms. The starting point in this work is Linux version 3.6 with the corresponding RT patch.

Overview Linux Development process Setting up LinuxRT is composed of many steps compared to FreeRTOS:

1. Configuring and building U-Boot (SSBL)
2. Configuring and building Linux-Image (wrapped with U-Boot header):
   a) Build driver for the new hardware device
   b) Configure kernel
   c) Apply RT Patch
3. Creating a device tree blob[1)]
4. Building a FSBL
5. Creating a boot image from the files produced in the previous steps
6. Configuring and building a file system

[Picture of Linux Design Flow]

[1)] A device tree is a data structure which describes the underlying hardware. A device tree is passed to the OS at boot time, so it can initialize the hardware dynamically.[2]

Configuring Linux The goal of the configuration is to create a minimal version of Linux. This means to eliminate unnecessary drivers and kernel tools (e.g. tracing tools) from the system. Driver initialization has a negative impact on the boot time. Moreover, unnecessary drivers can cause a larger memory print of the system and delays during runtime. As the detailed process of driver elimination is not of interest at this point, only the most important steps will be pointed out:
The OS under test is based on a real system which will be used by Siemens in later projects. Therefore, only the drivers needed in this system will be kept in the kernel: The most important ones are SPI, Ethernet and Universal Asynchronous Receiver Transmitter (UART) driver. All unnecessary drivers for Universal Serial Bus (USB) devices, blue ray or CD player and more were removed. Further, there are options which allow kernel tracing (e.g. ftrace) which were disabled when running the experiments. The final configuration reduces the kernel size from [??] MB to 1,9 MB.

3.1.3 Time measurement

The time measurement should be as accurate and fast as possible. High measuring overheads could falsify the results, so software timers should be avoided. One elegant way is to access the CPU cycle counter register of the ARM Cortex-A9 processor using inline-assembly code [Reference to the ARM Cortex A9 Architecture Manual]. This code is compiled into four assembly instructions when copying data into an array. The advantages of this method are not only a low overhead but also the OS independence of the assembly code. Hence this method can be used for both FreeRTOS and LinuxRT. However, the CPU has to keep running at the same frequency during the whole process otherwise the results will be falsified.

3.2 Boot time

The boot time is the time from powering on the hardware until the first (user) task is run. On the ZC702 Evaluation platform, this time can be measured by utilizing user Light Emitting Diodes (LEDs) on the board. The default state of those LEDs is on, so by turning them off in the first executed task, the exact power up time of the system can be measured. As already mentioned (see 2.4.4), the boot time depends on the complexity of the operation system and the hardware. Furthermore, the evaluation board provides three different ways of booting an OS:

1. Via Joint Test Action Group (JTag)
2. From Secure Digital (SD) card
3. From QSPI flash

The boot from QSPI flash is the fastest, hence it will be used in the following experiments. The following flow graph illustrates very well the differences between the boot process in

FreeRTOS and LinuxRT.

[Picture Comparison Boot Flow Linux and FreeRTOS]

3.2.1 Booting in FreeRTOS

The booting process of FreeRTOS is shown in the flow graph, all steps are executed. There is barely room for optimization in the first two steps of the booting process. However, the last two steps are dependent on the size of the bitstream and the size of the application respectively.

3.2.2 Booting in Linux

Obviously, the Linux booting process is more complex than the FreeRTOS one. There are three bottlenecks in this process:

- Loading the Image from QSPI to RAM
- Loading the file system from QSPI NOR flash to RAM
- Configuring the FPGA

The default Linux configuration provided by Xilinx does not include an FPGA bitstream and is run from SD card. This takes about 15 seconds. The transfer of the Linux image and the file system to RAM takes about 12 seconds. Besides, by adding a bitstream, this time increases by another 15 seconds.

The very first step to decrease this time has already been done by compressing the Linux kernel. The next step is to move the bitstream inside of the RAM image. In Linux, the PL can be configures from the OS by simply writing the bitstream to the file corresponding to the FPGA. This decreases the FPGA configuration to a range of milliseconds. Finally, the last step is to change the default file system to a file system that does not need to be loaded by the SSBL. A good option is to create a *UBI File System (UBIFS)* flash file system [12]. UBIFS is based on Unsorted Block Images (UBI) which itself works on Memory Technology Device (MTD) devices [11]. MTD is an abstraction layer for raw flash devices (no block devices!) in Linux. On top of that, the volume management system UBI takes care of mapping logical erase blocks to physical ones. On one hand, UBIFS has the advantage of fast mounting speed which is not dependent on the flash size. On the other hand, UBI initialization depends on the flash size, so the partition size should be kept as low as possible. In the current configuration the initialization of the flash image takes about 3 seconds. Nonetheless, an experimental new UBI feature called *fastmap* is available from Linux kernel 3.7, which allows the attachment of a UBI device in almost constant time [13]. One more advantage is, that UBIFS allows to store changes in the file system. This is not possible when using a ram disk image.

UBIFS needs to be enabled in the kernel. The initialization is completely removed from the boot loader, it is done completely on kernel start-up.

3.3 Task Switching Time

The task switching time is the time it takes for a task to start after an other task has finished or yielded the processor. An important note is, that the task switching time is not preemption time. The processor is given away voluntarily. Both OSes implement a yield function which allows the calling task to invoke the scheduler. This can be used in the experiment.

Two tasks of the same priority are set up, their composition is the same. Both perform a for-loop with the number of executions. The first instruction in the loop is the recording of the start time. Then a yield is performed what causes a context switch. The first instruction of the newly scheduled task is the recording of the end time. Obviously, the first start time is recorded twice because the two tasks are equal. Hence, the start time of the last test run is not valid. When the for-loop is executed 50000 times in each tasks, 100000 context switches are performed in total. An advantage of this method is that all data is collected without overhead which has to be subtracted afterwards.

After the last measurement, an evaluation task is started. It prints the results to console (via UART) so they can be recorded.

[Picture Execution Flow]

3.4 Interrupt Latency

Interrupt latency is defined as the time which passes from triggering an interrupt until the first instruction of the interrupt service routine. For this experiment, the assumption is made that devices in real application are often connected as GPIO. Therefore, the mentioned interrupt generator will be utilized. It has a 5ms period of which the positive pulse is 5 us and the remaining time is negative. The interrupt latency measured here does not correspond to the time the CPU is actually handling the interrupt (see figure [...]). Therefore, the time in this experiment will be measured using an oscilloscope, not the CPU clock cycle counter. The detailed execution flow is described in the following:

1. Start of measurement: Interrupt is generated by FPGA.
2. Interrupt is handled by GIC.
3. CPU executes ISR (depending on implementation)
4. Stop of measurement: LED is turned on by the ISR

The only difference between the two operation systems is step number three. To calculate the exact interrupt latency, the time it takes to turn on the LED has to be subtracted. Consequently, this requires a further experiment:
The LED will be toggled in a for-loop. As the hardware needs some time to proceed the request, another for-loop is executed inside. The time for the for-loop can be measured individually and be subtracted from the execution time.

[Picture interrupt flow]

Interrupt Measurement FreeRTOS

As described before (see 2.3.1), executing the ISR is absolutely independent from the OS. There are only two interrupts which need to be considered here: The timer interrupt (highest priority) and the GPIO interrupt. Consequently, the execution time is expected to be constant except in the cases where the GPIO interrupt is preempted by the timer tick.

Interrupt Measurement LinuxRT

There are more interrupts involved (e.g. Ethernet and UART) in the LinuxRT system than in FreeRTOS. Dependent on the application, there is the possibility to implement the ISR as a threaded interrupt (default in RT) or to execute the handler in interrupt context (see 2.5.2). Both alternatives shall be implemented. It is expected that the non-threaded handling is more predictive.

3.5 Preemption Time

3.6 Semaphore Shuffle Time

3.7 Message Passing Time

The message passing time is the time which is takes a task to receive a message sent by another task. Two tasks with equal priorities can be used for the test. The first task tries to read from a task queue, it is blocked until a message arrives. Then the second task is scheduled, sends a message via the message queue and yields immediately. [what kind of message?] Consequently, the first task is unblocked and retrieves the message sent. Now that the queue is empty, the test run can be repeated. The starting time is measured before a message is sent via the queue by the second task, the ending time after the first task receives the message. [Picture with exeecution flow]

3.8 Deadlock Breaking Time

The deadlock breaking time is the time which it takes for the OS to resolve priority inversion caused by low priority tasks holding a resource needed by a high-priority task. The experiment is set up similar to the given example for priority inheritance (for details refer to 2.5.3) and repeated 100000 times: Task A with the lowest priority starts executing. A takes a mutex and starts a second task B with higher priority than itself. B preempts the first task and starts task C with the highest priority. C tries to take the same mutex as the first task and is blocked. Consequently, the priority of the A is raised, it can finish execution (in the test scenario there is no work to be done), release the mutex and unblock C. The starting time is measured before C takes the mutex, the ending time after C returns from the Blocked state. To repeat the test, the original state of the tasks has to be restored. Therefore, C cancels task B, so it will not preempt A and suspends itself. Now A will be scheduled and the test can run again.

[Picture with execution flow]

4 Results

5 Conclusion and Outlook

Bibliography

[1] eCosCentric Limited: *eCOS*. http://ecos.sourceware.org/, 2008. 3

[2] Embedded Linux Wiki: *Device Trees*. http://elinux.org/Device_Trees, 2013. 16

[3] Gökhan Ugurel, Cüneyt F. Bazlamaçc: *Context switching time and memory footprint comparison of Xilkernel and µC/OS-II on MicroBlaze*. In *7th International Conference on Electrical and Electronics Engineering*, pages 62–65, December 2011. 3

[4] Jones, M. Tim: *Inside the Linux Boot Process*, May 2006. 11

[5] Jones, M. Tim: *Inside the Linux 2.6 Completely Fair Scheduler*, December 2009. 7

[6] Kar, Rabindra P.: *Implementing the Realstone Benchmark*. In *Dr. Dobb's Journal*, volume 15, pages 46–55, April 1990. 3, 4

[7] Love, Robert: *Linux Kernel Development*. Developer's Library, 3 edition, June 2010. 7

[8] Marco Cereia, Ivan Cibrario Bertolotti, Stefano Scanzio: *Performance evaluation of an EtherCAT master using Linux and the RT Patch*. In *IEEE International Symposium on Industrial Electronics*, pages 1748–1753, July 2010. 3

[9] Mastura D. Marieska, Paul G. Hariyanto, M. Firda Fauzan: *On performance of kernel based and embedded Real-Time Operating System: Benchmarking and analysis*. In *International Conference on Advanced Computer Science and Information System*, pages 401–406, December 2011. 3

[10] Micrium: *µCOS II - The Real-Time Kernel*. http://micrium.com/rtos/ucosii/overview/, 2013. 2

[11] MTD - Memory Technology Devices: *General MTD documantation*. http://www.linux-mtd.infradead.org/doc/general.html, 2011. 18

[12] MTD - Memory Technology Devices: *UBIFS - UBI File System*. http://www.linux-mtd.infradead.org/doc/ubifs.html, 2011. 18

[13] MTD - Memory Technology Devices: *UBI - Unsorted Block Images*. www.linux-mtd.infradead.org/doc/ubi.html, 2012. 18

[14] Politecnico di Milano - Dipartimento di Ingegneria Aerospaziale 2005 - 2006: *RTAI - the RealTime Application Interface for Linux from DIAPM*. https://www.rtai.org/, 2013. 2

[15] Rabindra P. Kar, Kent Porter: *Rhealstone: A Real-Time Benchmarking Proposal*. In *Dr. Dobb's Journal*, 1989. 3, 4

[16] Real Time Engineers Ltd.: *FreeRTOS*. http://www.freertos.org/, 2013. 2, 5, 6

[17] Steven Rostedt, Darren V. Hart: *Internals of the RT Patch*. In *Ottawa Linux Symposium*, volume 2, June 2007. 11, 12

[18] Tran Nguyen Bao Anh, Su-Lim Tan: *Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers*. In *Micro, IEEE*, August 2009. 3

[19] Wolfgang Betz, Marco Cereia, Ivan Cibrario Bertolotti: *Experimental Evaluation of the Linux RT Patch for Real-Time Applications*. In *IEEE Conference on Emerging Technologies and Factory Automation*, pages 1–4, September 2009. 3

[20] Xenomai: *Xenomai: Real-Time Framework for Linux*. http://www.xenomai.org, 2013. 2

[21] Xilinx Inc.: *UG708: XilKernel*. United States, October 2011. 3

[22] Xilinx Inc.: *UG585: Zynq-7000 All Programmable SoC Technical Reference Manual*. United States, 1.6 edition, June 2013. 15

[23] Xilinx Inc.: *UG850: ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide*. United States, 1.2 edition, April 2013. 15