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

EDF Earliest Deadline First

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

LIFO Last In First Out

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

RT Real-Time

RTAI Real-Time Application Interface

RTOS Real-Time Operating System

SD Secure Digital

SDK Software Development Kit

SMP Symmetric Multiprocessor

SPI Serial Peripheral Interface

SRTF Shortest Remaining Time First

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 two main classes, depending of the consequence caused by missing a deadline:

- Hard
- 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 soft real-time systems, deadlines can be missed but decrease the Quality of Service (QoS). These kind of systems are usually used for application with a continuous data flow like multimedia streaming applications or on-line reservation systems. Missing of one data packet in these application may cause noise in a video conference but no lives would be harmed.

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)* [16], *Xenomai* [24], 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* [12] or *FreeRTOS* [18]. 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* [8] [17]. This benchmark is based on six parameters (for details refer to 2.8.1) 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 [21]. 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 [5], memory footprint and context switching is compared between μCOS and XilKernel[1)] on the softcore processor Microblaze. Three different OSes - Xenomai, LinuxRT and eCos [3] - are compared in [11]. The performance metrics are similar to the Rhealstone benchmark. Depending on the application, eCos and LinuxRT perform better than Xenomai. The authors of [10] and [23] 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 [10] the focus is mainly on the effect of real-time priorities, in [23] 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 proposed by Kar and Porter [8] [17]. 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

[1)] XilKernel [25] is an RTOS developed by Xilinx.

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

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.

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.

2.1 The Scheduler

Every task or process created by the kernel or a user, is managed by the operation system - more precisely the scheduler. It manages the tasks and decides which task is allowed to operate next. 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 (fig. 2.1) used in FreeRTOS [18] is described. The state flow in Linux is related, but extended by other states which are not of interest for further consideration.

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

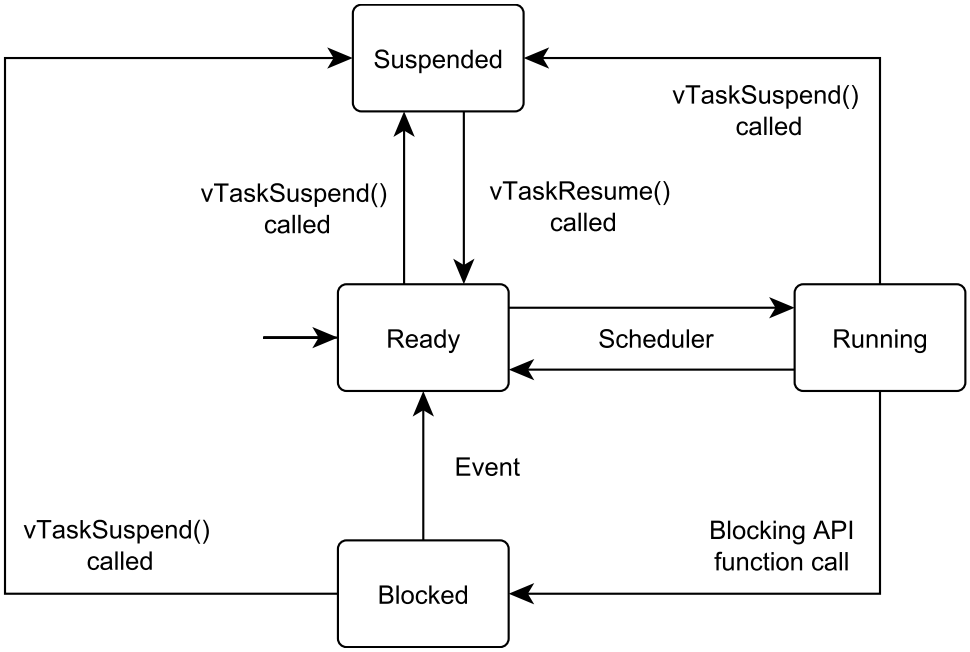

Figure 2.1: Task State Transitions in FreeRTOS with API Calls [18]

Blocked Tasks enter the *Blocked* state when they are waiting on a queue, 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. Its frequency 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 - the standard on both systems is 100 times. Therefore, processor time might be wasted. On the other hand, the OS overhead in the system grows if the tick period is too high. There are no instructions on how to determine the optimum tick period, this is dependent on the specific applications running on that system. Programmers can influence the scheduling by calling the *yield* function which invokes the scheduler. Moreover, the timer ISR is usually used to update the internal system time. When no high resolution timer is available, the granularity of the system time depends on the tick. As the timer tick is implemented in an ISR, it causes latency in the system by interrupting the currently executing code. Some OSes provide a configuration called *Tickless kernel* (see fig. 2.2). It disables the timer interrupt at all when there is no task runnable because rescheduling is not necessary.

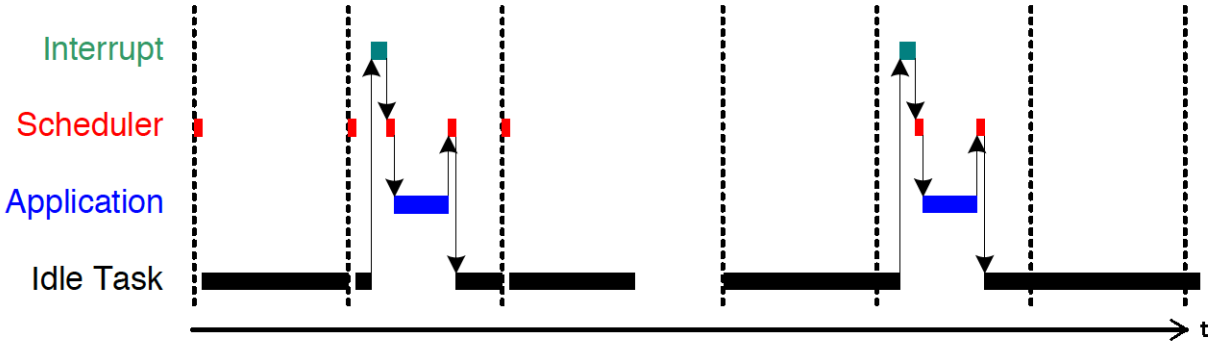

Figure 2.2: Standard tick (left), tickless kernel (right) [2]

2.1.3 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, e.g. scaling down frequency or executing the *halt* instruction. Moreover, it can be used to perform background processes or as indicator for capacity utilization.

2.1.4 Scheduling Policies

The scheduler decides which task will run next (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. Examples for cooperative scheduling are the *First-In-First-Out (FIFO)* or the *Last In First Out (LIFO)* algorithms. Round Robin (RR) or priority based scheduling (Earliest Deadline First (EDF) and Shortest Remaining Time First (SRTF)) are examples of preemptive scheduling. 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. For details on scheduling in Linux and FreeRTOS refer to the corresponding sections (2.6.2) and 2.5.1.

2.2 Intertask Communication and Synchronization

In OSes usually not only one but many different tasks are active. These tasks need ways to be synchronized with each other. Moreover, race conditions on critical resources, e.g. concurrent memory access, need to be prohibited.

2.2.1 Events

In many cases, tasks need to wait until another task completes execution. Instead of polling on 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. This kind of synchronization can also be realized using semaphores (see 2.2.3).

2.2.2 Barriers

Sometimes, multiple tasks have to be synchronized at one specific point in the execution flow. Therefore, all tasks have to wait until the others have finished. An example is the separation of matrix operations across multiple threads in a multicore system. Another example is an application which depends on different data sets collected by multiple threads. Before the tasks can continue their work, all data has to be available. Such synchronization points are called barriers.

2.2.3 Memory access

Concurrent memory access of multiple tasks can have undesired results as illustrated in the following example(see table 2.1) :
Person A and person B want to withdraw money from the same bank account (both 500 Euros) from a bank account via ATM at the same time.

Task A	Task B	Memory
–	–	X
get X	–	X
increment X by 500	–	X
–	get X	X
–	increment X by 500	X
write back X	–	X + 500
–	write back X	X + 500

Table 2.1: Example for concurrent memory access

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, so 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 A is scheduled. Task A writes back the new value X and finishes execution. Then, B also writes back its value X. Obviously, the newly written value of X - 500 Euros is wrong, it is supposed to be X - 1000 Euros.

To prevent such inconsistencies, the memory has to be locked by the accessing 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 limited to one.

2.2.4 Priority Inheritance

A problem which may occur while synchronizing memory access for tasks with different priorities is unbounded priority inversion. This means starving a high priority task by locking specific resources for an indefinite amount of time. An example is the following situation [20]: 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.

2.2.5 Message Passing

One way to exchange information between different tasks is passing messages between each other. Usually, one task blocks on a message queue until another task puts a message in this queue. This event wakes up the blocking task,so it can retrieve the message. The mechanism is a mix of signaling event and synchronizing shared memory access. It is available in Linux as well as in FreeRTOS.

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 hard drive controllers. The interrupt signal from a 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 loaded to be executed. Because they disturb the execution of other processes, ISRs should be kept as short as possible. Typically, only urgent operations are performed in an ISR. For less critical work, e.g.

copying data into the 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 enables interrupt nesting. This causes high priority interrupts to disturb the execution of lower prioritized interrupts.

2.4 Hardware Delays and OS jitter

Different delays caused by the underlying hardware or the OS are crucial in real-time applications. In this work, the term *delay* refers to unexpected events which extend the execution time of a real-time task.

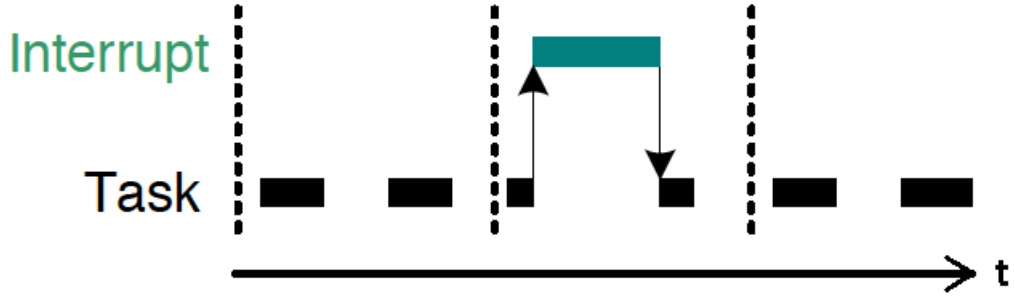

Figure 2.3: Delay in task caused by interrupt

In contrast, *latency* is the time a real-time task or a piece of code actually needs to execute. There are mainly three kind of delays and one kind of latency:

1. Delays from the Operation System
2. Delays from interrupts (fig. 2.3)
3. Delays from cache misses
4. Latencies from task execution and synchronization

Moreover, the boot time can be crucial in special real-time application.

2.4.1 Delays from the Operation System

The obvious kind of delay which is inevitable when using an OS is the delay caused by the OS itself. The most significant (and predictable) delay is caused by the timer tick. More delays can be produced by background daemons and other system services.

2.4.2 Delays from Interrupts

The second 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 mainly caused by I/O devices, the delays from interrupts will increase when the system heavily communicates with a large number of peripherals.

2.4.3 Delays from Cache Misses

The last cause of delays is caching. Caches are fast buffers which allow faster data access to data currently cached. They are often built up hierarchical (level 1, level 2, ...) and divided into data and instruction caches. A typical set up are separate level 1 data and instruction caches for each CPU and a shared level 2 cache for both data and instructions for all CPUs. On one hand, it caches increase the performance of an application significantly by reducing slower memory accesses to the RAM. On the other hand, the cache is used by every process on the system. This can cause data currently needed by a real-time application not to be present in the cache, a cache miss occurs. The reload of the data to the cache causes a delay.

2.4.4 Latencies from Task Execution and Synchronization

The last aspect is more correctly defined as latency, not as delay. It specifies the time which is caused by using several synchronization methods provided by the OS. These latencies are implementation specific. This time is predictable:
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 2.1.2), 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 rescheduling can be caused by giving a semaphore, sending a message or waking up the task from an ISR.

Besides, task synchronization causes latencies in a 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. If a task with higher priority has been woken up, the current task is preempted and a context switch takes place. Dependent on the application and the right synchronization mechanism, the overall system latency can be reduced.

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

2.4.5 Boot Time

The boot time can be a relevant factor for the choice of a specific OS if the system needs to start up very fast. When a car is started, the driver does not want to wait 15 seconds until the ignition starts running. Obviously, the boot time depends on the memory print of the operation system and on the medium from which the OS is loaded. 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 a First Stage Boot Loader (FSBL) is usually sufficient. This can as well optionally load an FPGA design. For more complex systems like Linux [6], 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 and a hardware initialization file into memory (device tree). After those steps, the kernel can be decompressed and finally boot. While booting, it initializes the file system with the previously loaded image as well as 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 FreeRTOS

This section described the FreeRTOS in more detail. The focus is on the scheduling algorithm, interrupt handling and synchronization features.

2.5.1 Scheduling in FreeRTOS

Scheduling in FreeRTOS [18] 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 (see fig. 2.4). 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 RR algorithm is used to pick the one.

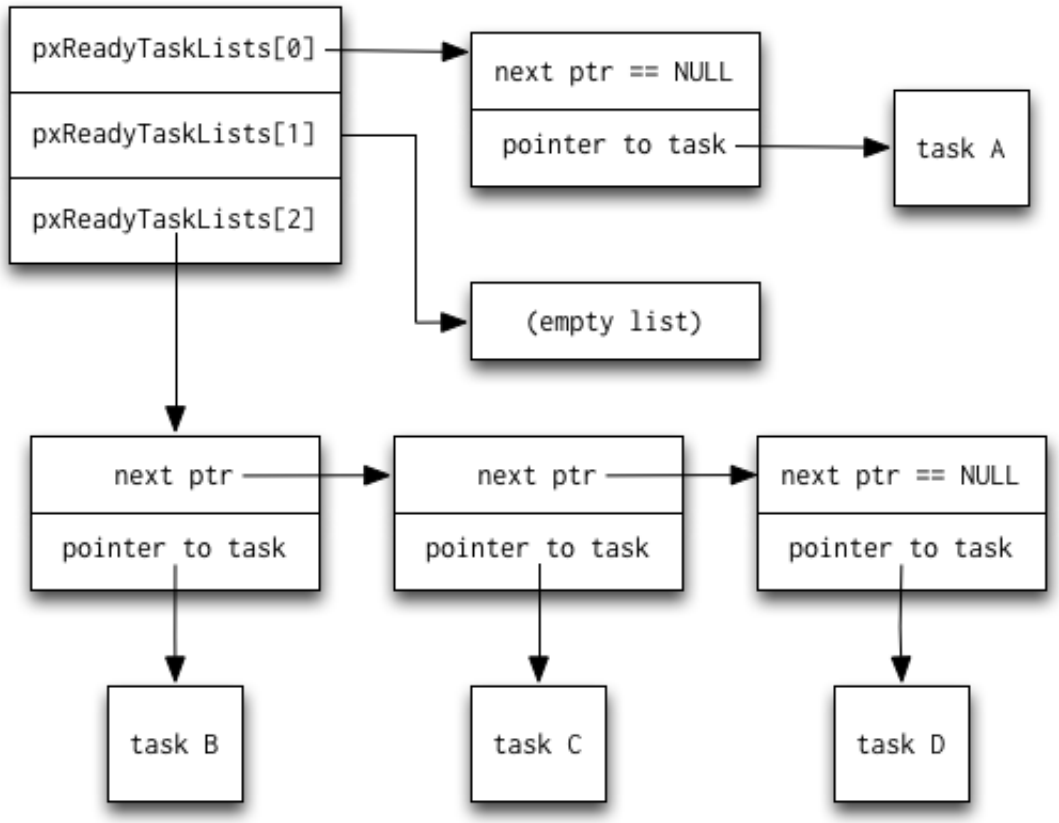

Figure 2.4: FreeRTOS Ready queue[**?**]

2.5.2 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.

2.5.3 Synchronization Features in FreeRTOS

In this section a short overview of the FreeRTOS Application Programming Interface (API) is given. Most API functions are implemented for the usage in tasks as well as in ISRs. ISR functions have the suffix *FromISR*.

Tasks and Co-routines

Every FreeRTOS application is structured as a set of tasks and/or co-routines. Each task is provided with its own stack where the current processor context is saved on a task switch. Tasks are fully prioritized, support full preemption and are not restricted in usage. Co-routines, on the other hand, share a stack within the same application. This reduces their amount of RAM needed but requires special considerations and restrictions in implementation (for more details refer to [19]).

Semaphores

FreeRTOS supports three kind of semaphores:

- Binary Semaphores
- Mutexes
- Counting Semaphores

All semaphores support priority inheritance. The implementation is based on message queues. Mutexes and counting semaphores correspond to the memory access synchronization tools mentioned above (see 2.2.3). These semaphores need to be given back once taken. Binary semaphores are used as signals for event synchronization. In contrary to counting semaphores and mutexes, binary semaphores are taken by the task which is waiting for an event and given by the signaling task.

2.5.4 Booting in FreeRTOS

The booting process of FreeRTOS consists of booting the Read Only Memory (ROM) file, loading and running the FSBL, optionally loading the bitstream and finally loading and running the application (compare to figure 3.1). 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.

2.6 LinuxRT

This section described the Linux OS in more detail. The focus is on memory management, the scheduling algorithm, interrupt handling and synchronization features. Moreover, the differences between the standard Linux Kernel and the RT patch.

2.6.1 Memory Management

The Linux OS uses a quite complex system Linux uses a file system to store user programs and system data. This file system is usually larger than the available RAM, therefore it is provided on an other medium. The medium can be either located on a removable flash disk like an Secure Digital (SD) card, a Quad SPI (QSPI) flash or another medium. Because it provides much faster data access, the RAM is much more expensive and usually smaller than the other medium. For faster data access, program data stored in the file system is loaded from hard disk into RAM on run time. This is called mapping physical memory (from the file system) to virtual memory (RAM). Because of efficiency considerations, the data blocks are managed in pages of consequent data which usually consist of several kBs. The pages used by the real-time application could be flushed from the RAM which causes hardly predicable delays by reloading of pages. Yet, there is the possibility to to lock specific pages for critical data or applications in RAM. This feature should be used carefully because it might increase the run time of one application but decrease the over-all system performance.

2.6.2 Scheduling in Linux

Scheduling in Linux [9] [7] 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 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.

2.6.3 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 by calling the function *irq_request()* and deregistered by calling *irq_free()*. When an interrupt occurs (compare figure 2.5), the *do_IRQ()* kernel function occurs which takes care of all ISRs. Interrupt handler cannot be implemented by a user application. Application run in user mode have to use read, write or I/O control functions to access the specific device.

2.6.4 Synchronization Features in Linux

Linux provides multiple APIs with a variety of task synchronization features. Portable Operating System Interface (POSIX) compliant features are most common because of their compatibility across many platforms.

Processes and Pthreads

A program which runs in Linux is called a process. Each process has an own process ID and maintains its own stack. The usual way to create multi-tasking systems in Linux is by using so called *pthreads*. pthreads are light-weight tasks which have their own stack but share the parent's virtual memory and environment. They can be prioritized and scheduled individually.

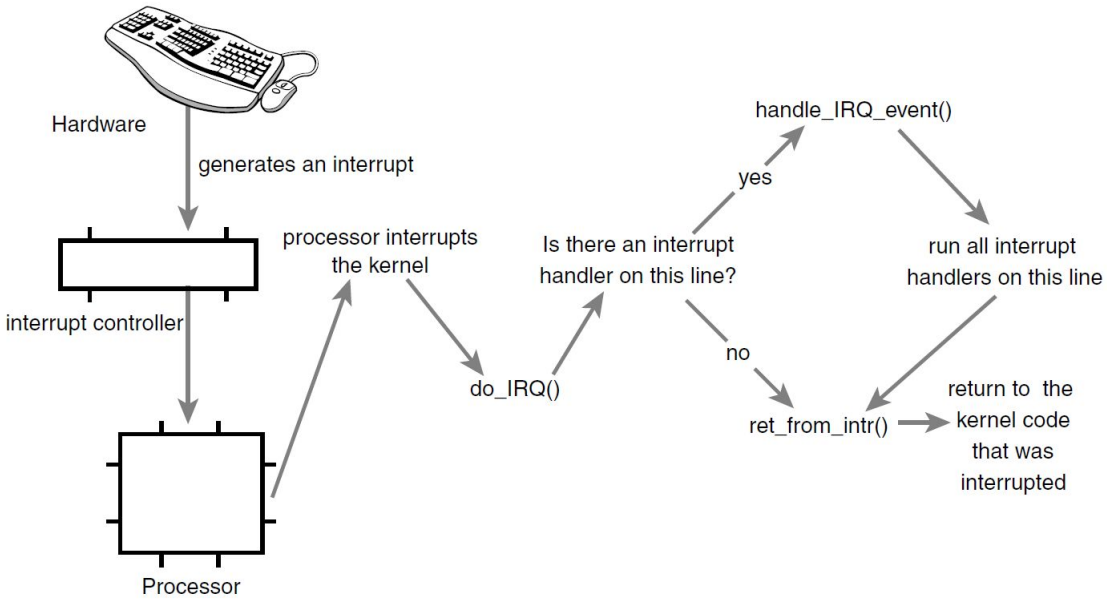

Figure 2.5: Path of an interrupt from hardware through the kernel [9]

Semaphores

In Linux, there are similar kind of semaphores compared to FreeRTOS. The POSIX API provides access to counting semaphores which are referred to simply as semaphores. Further, mutexes can be used on their own and in combination with condition variables. Condition variables provide a way to notify a specific thread when a special event has occurred. On initialization, attributes can be passed to semaphores and mutexes. The attribute determines for example whether a mutex supports priority inheritance or not.

Barriers

Barriers are a very convenient and powerful way of synchronizing threads. The API provides conventional barriers as described above (see 2.2.2) and the *join* function. The difference between the two is that the barrier synchronizes multiple running threads but the join function allows one thread to wait for the termination of other threads. Only one thread can call the join function for another thread, the result of multiple calls is undefined.

2.6.5 Booting in Linux

Obviously, the Linux booting process is more complex than the FreeRTOS one (see figure 3.2). Linux uses not only an FSBL but also a SSBL. Moreover, Linux needs a file system

to run properly and a device tree. The device tree has to be loaded during booting as well and provides information on hardware initialization. If an FPGA design is included in the system, this has to be loaded as well.

All these factors have an impact on the boot time and there are different configuration alternatives for example to initialize the FPGA or to load the file systems (see 3.2.2 for more details).

2.6.6 RT patch vs. standard kernel

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).

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.

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 2.6.2) 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.

2.7 Mathematical Quantification of Operation Systems

There are many different parameters to quantify RTOSes. Some important ones besides real-time capability are freedom of charge, memory footprint, available drivers, licensing, API richness and support [21]. Based on these categories, FreeRTOS and LinuxRT were decided to be the most promising candidates to use in future projects. To determine which one of the two is suitable for which kind of real-time applications, a set of tests has to be defined and performed. Consequently, the parameters which have an influence on the execution time have to be determined and analyzed. In the following, the boot time and the real-time task or application run time are described mathematically. This formula will later be used to evaluate the performance and the OS consideration of a test application.

2.7.1 Boot Time

The boot time t_{boot} can be calculated straight forward from the already discussed parts of the boot flow (refer to 2.7.1):

- Time to boot ROM t_{rom}
- Time to load the FSBL t_{fsbl}
- Time to program the FPGA t_{fpga}
- Time to load the OS t_{osload}
- Time to load the SSBLt_{ssbl}
- Time to load device tree t_{tree}
- Time to load the file system $t_{filesys}$
- Time to boot the OS t_{boot}

As already discussed, the complexity of the boot process is dependent on the OS. For Linux, the boot time consists of all steps listed above whereas FreeRTOS only needs the first three steps.

2.7.2 Applications and Real-Time Tasks

The execution time of a task is referred to as latency. It depends mainly on the executed application code, the underlying hardware and the latency caused by calls to synchronizing mechanisms of the OS. For real-time tasks, latency should be as deterministic as possible. Unexpected delays, mainly from interrupts, cause latency to vary. Moreover, unexpected delays can be caused by loading memory pages to RAM. This variation is called jitter and, logically, should be as low as possible.

Based the introduced delays in section 2.4, the execution time of a specific real-time task or application[1)] can be described by the following factors:

1. Deterministic part $t_{det} = t_{app} + t_{sync} + t_{isr}$:
   a) Execution time for application code t_{app}
   b) Execution time for OS related features where n_{os} is the number of OS related features and x^i_{sync} is the number of times this feature was used in the application $t_{sync} = \sum_{i=1}^{n_{sync}} x^i_{sync} * t^i_{sync}$
   c) Execution time for ISRs belonging to the application n_{isr} is the number of ISRs executed during the application and x^i_{isr} is the number of times this interrupt occurs during the application $t_{isr} = \sum_{i=1}^{n_{isr}} x^i_{isr} * t^i_{isr}$
2. Not deterministic part, jitter $t_{jitter} = t_{tick} + t_{int} + t_{cache} + t_{swap} + t_{daem}$:
   a) Delay caused by timer tick t_{tick}
   b) Delay caused by other interrupts where n_{int} is the number of interrupts available in the system $t_{int} = \sum_{i=1}^{n_{int}} t^i_{int}$
   c) Delay caused by cache misses where n_{cache} is the number of cache misses $t_{cache} = \sum_{i=1}^{n_{cache}} t^i_{cache}$
   d) Delay caused by unexpected memory remapping or swapping in the RAM t_{swap}
   e) Delay caused by OS daemon processes where n_{daem} is the number of daemon processes and x^i_{daem} is the number of times this daemon is run during application execution $t_{daem} = \sum_{i=1}^{n_{daem}} x^i_{daem} * t^i_{daem}$

t_{jitter} is also dependent on the probability of the single delays. As only the worst case execution time is of interest for further consideration, the formula stated above can be used for the calculation of t_{jitter} without modification.

2.8 Benchmark Preparation

To compare the two OSes and to decide which is suitable for a specific application, the parameters described in the previous section have to be determined. Only features available in both OSes will be considered. The goal is to determine worst case times for all of the single parameters and determine the influence of the jitter factors. The needed benchmarks for the deterministic and non-deterministic part will be briefly discussed in the following:

[1)] In the following, the term application also refers to one run of a critical real-time task.

2.8.1 Rhealstone Benchmark for deterministic Application Latency

The application code is not available at the time where the times are measured, so t_{app} has to be estimated. As the two OS are running on the same hardware platform, t_{app} is equal for both OSes.

For the latencies related to the OS and the interrupt latency the Rhealstone benchmark can be utilized. As already mentioned, it was introduced by Kar and Porter in 1989 [17] and was further modified and published by Kar in 1990 [8]. The purpose of the Rhealstone benchmark was to find a metric for real-time systems which is comparable to the Dhrystone [22] and Whetstone [1] benchmarks. This benchmark is a weighted mean of the following six parameters:

- Interrupt latency
- Task switching time
- Preemption time
- Semaphore shuffling time
- Deadlock breaking time
- Message passing latency

Details on the single parameters are given in the next chapter (see 3). In contrast to Kar's procedure, the absolute values are used for this work.

2.8.2 Jitter

The jitter t_{jitter} should be as low as possible. This requires careful consideration while configuring the underlying OS and designing real-time applications. Therefore, all causes of delays are briefly examined whether they can possibly be eliminated.

Timer Tick The timer tick occurs at a predefined frequency and cannot be eliminated without radical changes to the kernel. One benefit is that this delay is deterministic and can be included into the application design.

Interrupts Interrupts usually occur infrequently and hence cause undefined delays. In FreeRTOS, the only interrupt always available in the system is the timer tick. Whenever other interrupts are added to the system, they are not considered jitter. In LinuxRT, the number of interrupts depends on the drivers loaded. Yet, there is still the possibility to remove unnecessary drivers and control the inputs of the system to avoid as much jitter as possible.

Cache Misses Cache misses depend on the hardware and cannot be controlled actively. Careful application design can minimize the memory print and therefore the number of cache misses.

Page Swapping If the OS uses virtual memory space, page swapping can cause delays during the execution of a real-time task. To prevent this issue, Linux provides a function to lock specific memory pages in RAM. This function was specifically designed to be used to support deterministic behavior in real-time applications. Nevertheless, the programmer is in charge of using this function correctly.

Daemons Daemons are background processes which may not only cause jitter but also slow down the boot process. They do not exist in FreeRTOS. Unnecessary daemons in Linux can be detected and removed from the 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. The results will be discussed in a separate chapter.

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 [27] with an Zynq-7000 XC7Z020 [26]. 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 ?? kB 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 QSPI flash memory.

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

When the Zynq platform is used in future projects, most likely both PS and PL will be utilized. This chip is especially useful to combine a powerful CPU and own hardware designs. Therefore, an interrupt generator was implemented in the PL part to utilize the PL. It is connected as GPIO device.

3.1.2 Development Tools

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).

3.1.3 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 FreeRTOS kernel. Therefore, the OS itself does not need to be configured. Still, the hardware needs to be initialized appropriately on system startup (refer to 2.5.2) and an ISR needs to be implemented. 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 own distributions of embedded Linux which can be compiled for many different platforms. The starting point in this thesis is the Xilinx Linux version 3.6 with the corresponding RT patch. This was the most recent version available at the beginning of this work.

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)]

[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.[4]

4. Building a FSBL
5. Creating a boot image (*zlib* compressed) from the files produced in the previous steps
6. Configuring and building a file system

[Picture of Linux Design Flow]

Configuring Linux The goal of the configuration is to create a minimal version of Linux. This means to eliminate kernel tools (e.g. tracing tools) and unnecessary drivers from the system. Driver initialization has also a negative impact on the boot time. Moreover, 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. This does not necessarily have an impact on the implemented test application: The most important drivers are SPI, Ethernet and Universal Asynchronous Receiver Transmitter (UART) driver. All other drivers for Universal Serial Bus (USB) devices, blue ray or CD player, and more will be removed. Further, there are options which allow kernel tracing (e.g. ftrace) which will be disabled when running the experiments. The final configuration reduces the kernel size from 2,8[?] MB to 1,9 MB.

3.1.4 Time measurement

The time measurement should be as accurate 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 the register value to an array. The advantages of this method is not only 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 measurement process, otherwise the results will be wrong.

3.1.5 Application Design

This section provides a short overview over the general application design and execution. To prevent interruption by other tasks, the real-time tasks must have the highest priority in the system. Usually, 100000 measurements are collected per operation system per test. If this is not valid for a special case, it will be mentioned in the benchmark description.

FreeRTOS

Generally, there are no other tasks in FreeRTOS than the ones created for the test. Consequently, the real-time tasks have the highest priority in the system anyway.

LinuxRT

In Linux, multi-tasking applications are realized using pthreads. The development of an application should be kept as simple as possible, therefore all applications will be implemented in user space. All real-time thread in Linux get the highest possible priority using the preemptive RR Real-Time (RT) scheduling class. As the parent process and its child threads are scheduled individually, the parent has to have a higher priority than the children while creating the child threads. If not, it will be interrupted by the first thread with higher priority and hence cannot invoke the other ones. When all threads have been started, the priority of the parent is simply decreased below the one of the lowest real-time thread.

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. By turning them off in the first executed task, the exact power up time of the system can be measured. As already mentioned (see 2.7.1), the boot time depends on the complexity of the operation system and the hardware. Further, the evaluation board provides three different ways of booting an OS:

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

The boot from QSPI flash is the fastest and the one most likely used in later projects. Hence it will be used in the following experiments as well. The flow graphs illustrate the differences between the boot process in FreeRTOS (fig. 3.1) and LinuxRT (fig. 3.2.2 and fig. 3.2.2).

3.2.1 Booting in FreeRTOS

The booting process is already described in the previous chapter (see 2.5.4). Based on the formula derived in section 2.7.1, the FreeRTOS boot time can be quantified as follows:

$t_{boot}^{free} = t_{rom} + t_{fsbl} + t_{fpga} + t_{osload}$

Whereas there is no time optimization needed for FreeRTOS because the complete boot takes only 1,1 seconds, there are multiple optimization possibilities for LinuxRT.

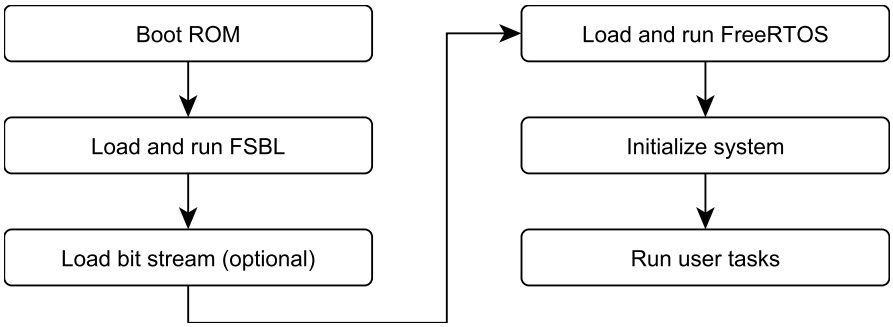

Figure 3.1: FreeRTOS Boot [28]

3.2.2 Booting in Linux

The booting process in Linux is shown in figure 3.2.2, it can be quantified as follows:

$t_{boot}^{Linux} = t_{rom} + t_{fsbl} + t_{fpga} + t_{osload} + t_{ssbl} + t_{tree} + t_{filesys} + t_{boot}$

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 15 seconds in total. Whereas t_{boot} is only 3 seconds and t_{fpga} is 0, the other times sum up to 12 seconds, which is mostly due to the transfer of the Linux image and the file system . Besides, by adding a bitstream (copying it to the SD card), this time increases by another t_{fpga} of 15 seconds.

The very first step to decrease this time has already been done by compressing the Linux kernel (see 3.1.3). The next step is to move the bitstream inside of the boot image. This has to be done not only to speed up the booting process but also because of the limited QSPI size of 128 kB. By moving the bitstream into the boot image, it is compressed as well. In Linux, the PL can be configured from the OS by simply writing the bitstream to the device file corresponding to the FPGA. This decreases t_{fpga} 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 [14]. UBIFS is based on Unsorted Block Images (UBI) which itself works on Memory Technology Device (MTD) devices [13]. MTD is an abstraction layer for raw flash devices (no block devices!) in Linux. On top of that, the volume management system

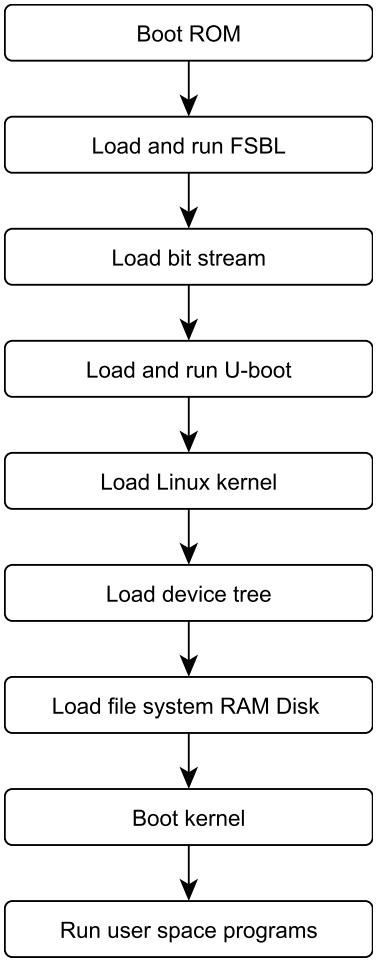

Figure 3.2: Xilinx Standard Boot [28]

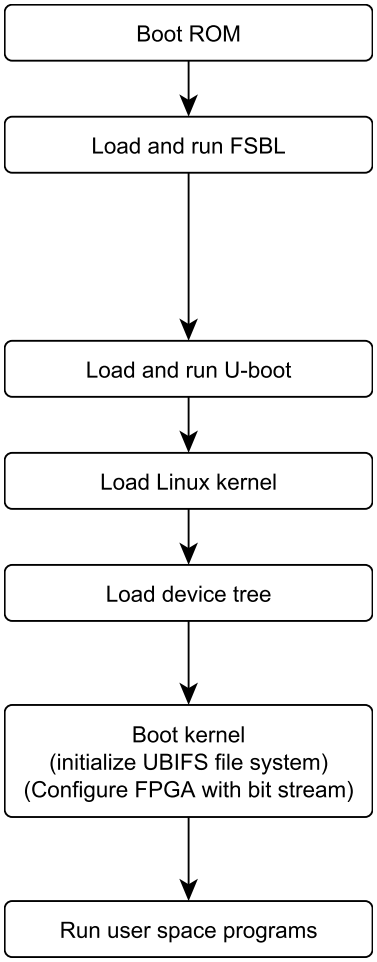

Figure 3.3: Xilinx Custom Boot [28]

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 time of the file system $t_{filesys}$ takes 3 seconds and is part of t_{osload}. 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 [15]. Another advantage is, that UBIFS allows changes to the file system to be stored at power-off. This is not possible when using a standard ram disk image.

The initialization is completely removed from the boot loader, it is done on kernel start-up. A disadvantage of this method is that UBIFS needs a reconfiguration of the kernel.

3.3 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 described 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 **??**). 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]

3.3.1 Interrupt Measurement FreeRTOS

As described before (see 2.5.2), 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.

3.3.2 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 2.6.6). Both alternatives will be implemented. It is expected that the non-threaded handling is more predictive.

3.4 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. It is important to point out, that the task switching time is not preemption time because the processor is given away voluntarily. Both OSes implement a yield function which allows the calling task to invoke the scheduler. It 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 task, 100000 context switches are performed in total. An advantage of this method is that all data is collected without any overhead which has to be subtracted afterwards.

[Picture Execution Flow]

3.5 Preemption Time

The preemption time is the time which it takes to interrupt a task and schedule another task of higher priority. This can happen when a task with a high priority is woken up by an other task or ISR triggering a signal or releasing a mutex. In this experiment, there is one low priority task A and a high priority task B. B starts first and blocks on a signal event. A is running in a permanent loop and reading the current CPU cycle counter register. At some point in time, an IRQ occurs and the task is interrupted. From the ISR, the signal is triggered to wake up B, consequently, a context switch happens. When B is activated, it measures the ending time and yields. This reactivates A and the test starts over. [diagram]

3.6 Semaphore Shuffle Time

The semaphore shuffle time defined by Kar is used to measure the overhead caused by using semaphores to ensure mutual exclusion. More precisely, it is the time between a task B being blocked on a semaphore which is taken by task A and and the unblock when the semaphore is released. Based on the tests, mutexes are used to implement mutual exclusion. Besides, it is possible, that a task protects a critical section by mutexes, but is not interrupted by another task. This case has to be considered as well. Further, signaling is a widely used feature to synchronize events, therefore it has to be included in the benchmark as well. Consequently, the semaphore shuffling results in three different tests.

Semaphore Shuffling Time by Kar In the first benchmark, two task are passing a mutex from one to the other. Task A starts by taking the the mutex and yields. Then the time measurement is started, task B tries to take the same mutex and is blocked. Now A is rescheduled, releases the semaphore and yields again. Task B is unblocked, the time is stopped and B releases the semaphore. By yielding the processor, task A is scheduled and the test starts over.

[diagram]

Semaphore in a single Task This benchmark is very easy to implement. A task requires a semaphore and releases it immediately after. The time it takes to perform these executions is measured. Then the test starts over.

[diagram]

Event Signaling This benchmark measures the time it takes a task to wake up on a signal event. Task A blocks on a signal. After that, task B is scheduled, triggers the signal and yields the processor. The trigger causes task A to wake up. The starting point of the measurement is before task B triggers the signal and the ending point after the awakaning of A. This sequence is repeated.

[diagram]

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 execution 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 2.2.4) 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] B.A. Wichmann, H.J. Curnow: *A Synthetic Benchmark*. In *Computer Journal*, volume 19, pages 43–49, February 1976. 21

[2] Barry, Richard: *FreeRTOS's Tick Suppression Saves Power*, 2013. 7

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

[4] Embedded Linux Wiki: *Device Trees*. http://elinux.org/Device_Trees, 2013. 24

[5] 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

[6] Jones, M. Tim: *Inside the Linux Boot Process*, May 2006. 12

[7] Jones, M. Tim: *Inside the Linux 2.6 Completely Fair Scheduler*, December 2009. 15

[8] Kar, Rabindra P.: *Implementing the Realstone Benchmark*. In *Dr. Dobb's Journal*, volume 15, pages 46–55, April 1990. 3, 21

[9] Love, Robert: *Linux Kernel Development*. Developer's Library, 3 edition, June 2010. 15, 17

[10] 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

[11] 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

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

[13] MTD - Memory Technology Devices: *General MTD documantation*. http://www.linux-mtd.infradead.org/doc/general.html, 2011. 27

[14] MTD - Memory Technology Devices: *UBIFS - UBI File System*. http://www.linux-mtd.infradead.org/doc/ubifs.html, 2011. 27

[15] MTD - Memory Technology Devices: *UBI - Unsorted Block Images*. www.linux-mtd.infradead.org/doc/ubi.html, 2012. 28

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

[17] Rabindra P. Kar, Kent Porter: *Rhealstone: A Real-Time Benchmarking Proposal*. In *Dr. Dobb's Journal*, 1989. 3, 21

[18] Real Time Engineers Ltd.: *FreeRTOS*. http://www.freertos.org/, 2013. 2, 5, 6, 12

[19] Real Time Engineers Ltd.: *Tasks and Co-routines*. http://www.freertos.org/taskandcr.html, 2013. 14

[20] Steven Rostedt, Darren V. Hart: *Internals of the RT Patch*. In *Ottawa Linux Symposium*, volume 2, June 2007. 9

[21] 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

[22] Weicker, Reinhold P.: *Dhrystone: A Synthetic Systems Programming Benchmark*. In *Communications of the ACM 27*, pages 1013–1030, October 1984. 21

[23] 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

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

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

[26] Xilinx Inc.: *UG585: Zynq-7000 All Programmable SoC Technical Reference Manual*. United States, 1.6 edition, June 2013. 23

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

[28] Xilinx Inc.: *ZC702 Boot from Flash*. http://www.wiki.xilinx.com/Zc702+Boot+From+Flash, 2013. 27, 28