T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE FOR EMBEDDED SYSTEMS

Project Number 288008

D 2.4 Design and Implementation of Interrupt Virtualization Hardware and APIs

Version 1.1
14 March 2013
Final

Public Distribution

Eindhoven University of Technology,Technical University of Denmark

Project Partners: **AbsInt Angewandte Informatik**, **Eindhoven University of Technology**, **GMVIS Skysoft**, **Intecs**, **Technical University of Denmark**, **The Open Group**, **University of York**, **Vienna University of Technology**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the Partners accept no liability for any error or omission in the same.

© 2013 Copyright in this document remains vested in the T-CREST Project Partners.

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft João Baptista Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: joao.baptista@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail: silvia.mazzini@intecs.it
Technical University of Denmark Martin Schoeberl Richard Petersens Plads 2800 Lyngby, Denmark Tel: +45 45 25 37 43 Fax: +45 45 93 00 74 E-mail: masca@imm.dtu.dk	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail: s.hansen@opengroup.org
University of York Neil Audsley Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325 500 E-mail: Neil.Audsley@cs.york.ac.uk	**Vienna University of Technology** Peter Puschner Treitlstrasse 3 1040 Vienna, Austria Tel: +43 1 58801 18227 Fax: +43 1 58801 918227 E-mail: peter@vmars.tuwien.ac.at

Contents

1 Introduction **2**

2 OS and Application Specifications **5**

2.1 OS and application description 5

2.2 OS and application behaviors 6

2.2.1 OS keeps a constant OS slot timing 6

2.2.2 Application behaviors are independent of interrupts 6

3 State of the Art **7**

4 Functions of the ICTM **8**

4.1 Overview 8

4.1.1 Specifications and Features: 8

4.1.2 Diagram 8

4.1.3 Supported functions 8

4.1.4 Reset strategy 9

4.2 Error cases 9

4.3 Clock counter 10

4.3.1 Signal description 10

4.3.2 Timing diagrams 12

4.4 Interrupt manager 13

4.4.1 Signal description 13

4.4.2 Timing diagrams 15

5 APIs of the ICTM **16**

5.1 Low level guidelines 16

5.2 Interrupt management 16

5.3 Clock counters 17

5.4 Advanced clock counters 17

5.5 OS functions 18

5.6 Operation Codes 18

5.7 Timings for the ICTM 20

6 Requirements **22**

7 Conclusions 24

Document Control

Version	Status	Date
0.1	First draft for review	18 February 2013
1.0	Final version	28 February 2013
1.1	Update on general requirement	14 March 2013

Executive Summary

This document describes the deliverable *D 2.4 Design and Implementation of Interrupt Virtualization Hardware and APIs* of work package 2 of the T-CREST project, due 18 months after project start as stated in the Description of Work. This document presents the design and implementation of virtualized interrupts and the interaction with scheduling of applications by real-time operating system (RTOS) and of scheduling of tasks by applications.

1 Introduction

The state-of-the-art real-time systems often contain multiple distributed applications with resource sharing on the computing platforms. Usually, these application can have a mixed time-criticality, i.e. a mix between firm-real-time (FRT), soft-real-time (SRT), and non-real-time (NRT) requirements. The co-existence of non-real-time tasks with real-time tasks mean that the worst-case response times of the latter might be affected, and perhaps even become unbounded. As a result, predictable preemptive schedulers, as illustrated in Figure 1, are required when multiple real-time and non-real-time applications are supported in the same system, to make sure the timing behaviors of multiple applications hosted on the same hardware platforms can never influence each other.

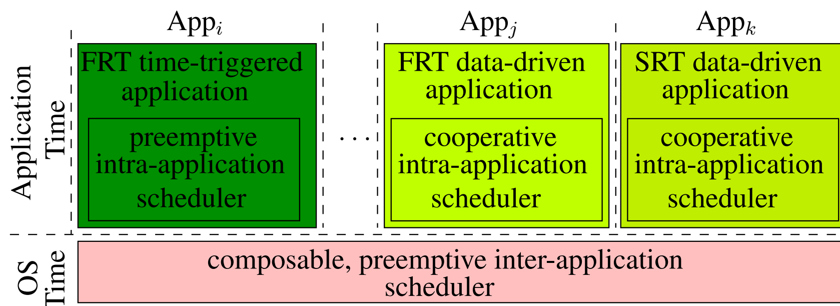

Figure 1: Composable, preemptive inter-application scheduler and the coexistence of preemptive and cooperative intra-application schedulers.

In the T-CREST system, a real-time operating system (RTOS) allows multiple applications to share a single processor, and allocates space (virtual memory addresses) and time (percentage of the processor usage) to each of the applications. In the remainder, we focus on the time-sharing or scheduling of applications on a processor.

In a real-time application, tasks may interact with the physical environment, besides communicating amongst themselves. Asynchronous inputs are generally obtained by the processor by receiving interrupts from the I/O device. However, since the interrupt service routine (ISR) consumes some of the execution time, an interrupt interferes with the execution and timing of an application. Furthermore, handling an interrupt that is destined for other applications has an impact on the current application as well, which makes different application to interfere with each other. Virtualizing I/O interrupts such that they are contained in the virtual platform belonging to the application is therefore required.

In the two-level scheduling mechanism, there are two logical interrupts used:

- One used by the inter-application scheduler to implement partitioning
- The other used for preemptive intra-application scheduling

To distinguish them, an interrupt vector is used to multiplex these two distinct logical interrupts on one physical interrupt line.

As the main contribution, we design and implement our interrupt virtualization mechanisms to handle these different interrupts based on an well-designed interrupt, clock, and timing manager (ICTM)

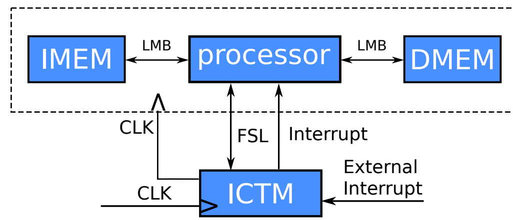

Figure 2: Hardware architecture of the ICTM and processor tile.

module. The hardware architecture of the ICTM interrupt system and the processor tile is schematically shown in Figure 2. External interrupts are received by the centralized ICTM controller from external cores and its own timer and performed through the virtualization control logic. Although the ICTM is currently being implemented on Xilinx FPGA with MicoBlaze core, the design of the ICTM is platform independent and can be used for other FPGA and processor technologies, e.g. Altera FPGA and Patmos processor [10].

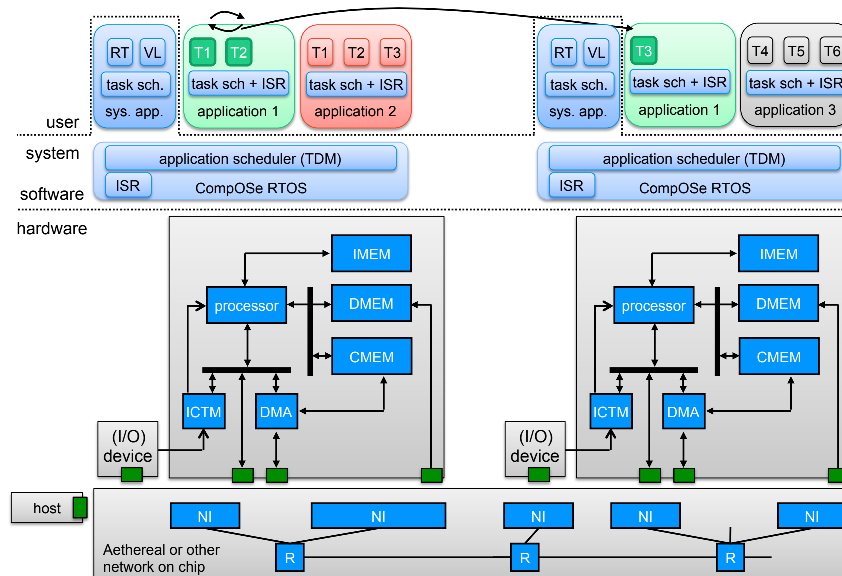

Figure 3: Illustration of virtualised interrupt structure with an ICTM in the processor tile and application-level ISRs in the RTOS.

The virtualized interrupt structure, with an ICTM in the processor tile and application-level ISRs in the RTOS, is illustrated in Figure 3. The ICTM accepts, generates, and routes interrupts on the basis of currently active application, and absolute or virtualized time of the processor. The functions of the ICTM should be consistent at any time and should not raise an error if things happen normally. Keeping the OS slot timings and values in hardware makes sure the timings of the whole system

stay exactly the same, which increases robustness. Since the OS slot is kept, it is assured that the application slot will never influence the OS slot.

The rest of this document is structured as follows. Section 2 introduces the OS and application specifications. Section 3 reviews the state of the art in interrupt virtualization in terms of characteristics and requirements. Section 4 then introduces the functions of the ICTM. The driver and APIs the ICTM are presented in Section 5. The general requirments are stated in Section 6. Lastly, conclusions are drawn in Section 7.

2 OS and Application Specifications

When multiple concurrent applications are mapped on the multi-processor computing platform, a set of communicating tasks, which potentially belong to different applications, may be mapped on one processor. To ensure the worst-case and actual timing behaviors of any application are not affected by the absence or behavior of any other application, the RTOS must partition applications, i.e. a preemptive time-division multiplexing (TDM) inter-application scheduler is needed. For each application, an individual non-preemptive (cooperative) or preemptive task scheduler, specific to its model of computation, must be specified on each processor on which the tasks are mapped.

There are two logical timers in the application time:

- One that keeps track of the current application slot and issues *application interrupts*;
- The other that keeps track of the task time and issues *task interrupts*.

A couple of issues on interrupt virtualization are to be addressed:

- How and where in the architecture are interrupts for applications other than the currently executing interrupt handled?
- How to ensure partitioning, such that no matter when any (set of) interrupts arrive it only affects the timing of the relevant application?

2.1 OS and application description

The OS slot consists of three parts: a part which finds the source of the interrupt and gates the system until an absolute time, acknowledges it and saves the registers, a part which saves the timers, run the application scheduler and load and save the timer values, and a part which loads the registers of the new application.

1. The first part is determining the source of the interrupt, acknowledging the interrupt and saving the context registers. After this the tile is gated until a given absolute time. This absolute time is programmed, and is computed by adding the *Jitterbound* constant with the last interrupt time. The variable part should be shorter than the *Jitterbound* value.
2. The second part is saving all the timer values, running the application scheduler and loading the timer values of the new application. After this the tile is gated until just before the third part. This gating makes sure the OS slot is exactly as long as programmed. The tile will be waked up by the ICTM just before the last part, and if the processor was not finished when it should have been waken up it generates an error as the period is not maintained since the context loading takes longer than expected which uses cycles of the application slot.
3. The third part is changing to the task stack of the new application and load the registers of the task. This loading takes a fixed time, and waking the tile up exactly as many cycles as this takes before the end of the OS slot makes sure the processor is just finished with loading when the application slot starts.

The application timer keeps increasing until the interrupt is acknowledged. This makes sure the values read by the application are still correct when reading in a critical region. However the application has stopped a certain amount of cycles before the interrupt is acknowledged. Therefore the application value should be subtracted by the amount of cycles between the last application cycle and the interrupt acknowledge. Then the application counter will have the same value as when the application was interrupted. The timings of the OS slot are illustrated in Figure 4. It shows that the OS slot and period are programmed, which implies the length of the application slot.

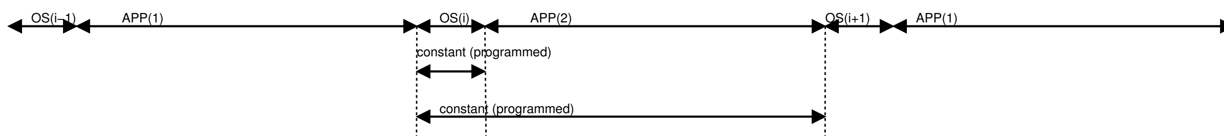

Figure 4: Diagram of the timings of periods

2.2 OS and application behaviors

The high level behaviors come from the OS and application level, which determine how the ICTM should be designed and implemented.

2.2.1 OS keeps a constant OS slot timing

The OS slot is responsible for managing and scheduling the applications. To make sure this functionality is maintained, the timings of the OS slot should be the same. This means it should first acknowledge the interrupt, then gate until a specified time for jitter removal, then load the application state (except for the registers), then gate for the remaining OS slot, and just before the application slot load the registers and jump to the application. The ICTM will enable the tile automatically exactly long enough to load the registers and jump to the application if the correct loading context value was programmed. This jump to the application should always be the last instruction before the actual application slot starts, so the application gets the complete time as was scheduled since the first instruction of the application slot is the first instruction resumed of the application. This first instruction is including filling the pipeline, so also the processor is in the exact state as when it got interrupted.

2.2.2 Application behaviors are independent of interrupts

This means the complete current state should be saved when interrupted and loaded when scheduled. The states which are saved and loaded are: registers, stack pointers, application counter, and future application interrupt(both in application time space as in system time space). The hardest of these states to maintain is the application counter, as it should be cycle accurate with respect to the uninterrupted application.

3 State of the Art

Interrupt virtualization has been a mechanism for handling either hardware- or software-initiated interrupts. In a virtualized environment, an interrupt manager is usually used to handle interrupt requests that would not be appropriately served by multiple guest RTOSs . However, virtualization only provides a functional isolation and can still lead to resource contention between different guest systems [5]. On a commercial mobile environment, OKL4 microvisor-based virtualization has been demonstrated [8], on which two virtual machines may interact via message-passing or shared memory. That is to say, temporal dependencies between different guest systems may still exist, which breaks the required composability in the T-CREST project.

For RTOSs conform to the ARINC standard [4], e.g. PikeOS [3], INTEGRITY [1], and LynxOS-178 [2], they follow a specification for time and space partitioning. Processing time is divided into number of time slices known as partitions, and every partition is assigned an application. To the best of our knowledge, cycle-level isolation between the isolated partitions, and subsequently, between the applications, has not yet been addressed. That is, a small partition jitter (the deviation between the reference time and the actual time) may still exist in ARINC systems.

With the support of two or more independent program counters in the processor hardware, multi-threaded processors have been proposed with a trivial thread-switch overhead [11]. That is, different threads of control can run in parallel within the processor pipeline to optimize the throughput of multiple workloads. In another work, with hardware supported partitioning mechanisms, performance counter is used to collect the performance information and determine the runtime behavior of different applications [9]. However, in either of these two approaches, different time-interleaved threads or applications may still interfere each other.

Based on the state of the art and requirements, we propose our ICTM module for interrupt virtualization to achieve cycle-level partitioning (a.k.a. composability) in the T-CREST project. A hardware interrupt management scheme is designed and responsible for providing a virtual interrupt mechanism to application partitions. In each application partition, all the interrupts belongs to other partition are blocked, which is also called *interrupt masking*. The system preempts the execution of other applications or tasks on guest partition via hardware-based transition mechanism. The composable computing systems differ from traditional systems by hiding the interrupt processing differences from virtualized partitions, and implement virtual interrupts with preemptive two-level scheduling.

4 Functions of the ICTM

The ICTM keeps track of the timers and manages the interrupts. These functions should be consistent at any time and should not raise an error if things happen normally.

4.1 Overview

The system is considered composable only if all the programs do not affect each other in any way, so the time slices must be exactly the same. Therefore periodic interrupts must be generated to guarantee this. The processor is able to program the ICTM so it will guarantee to generate periodic interrupts.

4.1.1 Specifications and Features:

The ICTM must comply to the following specifications:

- Two timers
- 64 bit timers
- All counters counting up
- One data bus
- Support for external interrupts

The new features of the ICTM are:

- Two 64 bit timers: one for system time and one for application time.
- Multiple interrupts support, including external interrupts.
- Automatic periodic interrupts.
- Able to MASK interrupts for certain applications.
- Works with functions: only one FSL bus needed.

4.1.2 Diagram

A diagram of the ICTM is shown in Figure 5.

4.1.3 Supported functions

A list of the supported functions for the ICTM are:

- **Interrupt management**: Ack int, read int reg, and set mask
- **Standard control functions for clock counters**: set/read, reset, start, stop, and init
- **Extra counter functions**: set period, and enable/disable period

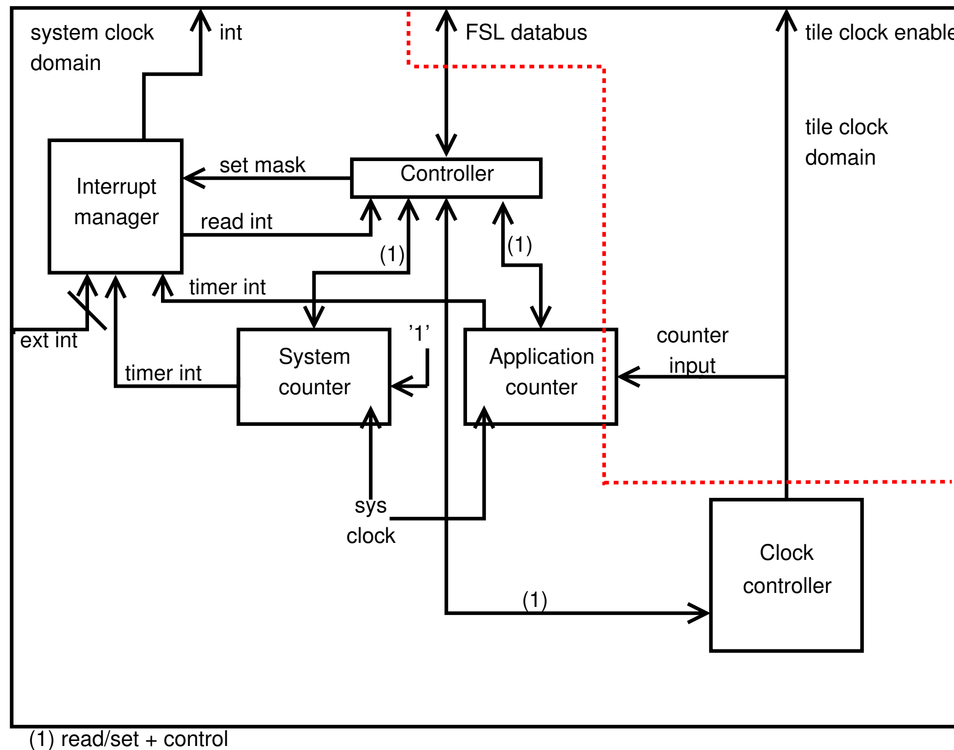

Figure 5: Diagram overview of the ICTM .

4.1.4 Reset strategy

The ICTM is responsible for the clock for the processor. Therefore it is important that the correct procedure is done before the processor starts to execute its programs. The global system reset module however already takes this into account: first the system bus reset are deasserted including the ICTM , and 16 cycles later the peripherals, and again 16 cycles later the processor. This gives enough time for the memory and the ICTM to be fully initialized.

When the system is reset both the counters are set to zero and are stopped, so the OS must first set and start the timers before running applications. This makes sure that there is no interrupt generated at the initialization steps of the OS.

4.2 Error cases

In the ICTM there are error cases possible in which the ICTM does not work as intended. These errors are registered in an internal register. Every period this register is checked and an error is reported if this register is not equal to zero. In this error the register value is transferred in a hexidecimal format for reference which error is registered.

Figure 6: Error register format

The numbers in this register, as illustrated in Figure 6, mean:

- 0: System timer is set in the past
- 1: Application timer is set in the past
- 2: System interrupt is not acknowledged in the corresponding OS slot
- 3: Denominator has been given a value of 0
- 4: Application preload length is equal to 0 (which means it is unset)
- 5: The end of the OS slot is reached while the system is not in the OS slot
- 6: An unset interrupt has been acknowledged
- 7: An interrupt is not acknowledged while it is raised again (interrupt raised while the previous has not been acknowledged)

4.3 Clock counter

The clock counter is responsible for generating the interrupts at certain moments specified by the processor. These interrupts are used for determining if a certain time slice is finished and a context switch should be done. This can be done by setting the interrupt time to a certain point in the future. When the counter has reached this moment the interrupt is generated so a context switch will occur. Since the data bus is 32 bits wide and the counters are 64 bits wide only half of the timer value can be read at the same time. To be certain the correct value is read out the second part which has to be send is stored at an internal register and the second half ready bit is asserted. If a second read is requested the second half of the value is send and the second half ready bit is deasserted.

4.3.1 Signal description

All the signals can be found in Table 1.

Table 1: Signal listing for the Clock counter

Signal name	I/O	Description
clk_control	I	Clock input for the control part of the counter. This clock is used for setting and reading the counter value.
clk_counter	I	Clock input for the counter. This clock is counted and can be different from the clk_control. In the current design it is given that the phase of the two clocks are the same
rst	I	Reset signal for the counter. This signal is active high (as is the rest). This will reset all the temporary values to 0 and stops the counter.
enable	I	The counter counts the clk_counter clock if enable is high, else keep current value.
read	I	If high, the highest half of the current value is put on the value_out lines, the lower half is stored in a register (so this will not change during the second half readout) which will raise the second_half_ready bit. If the second_half_ready bit is high read the second half of the time and lower the second_half_ready pin.
write	I	If high, the current counter value is changed to the value given in the value_in lines.
timer_write	I	If high when write is high, instead of changing the counter value the interrupt timer is set.
value_in [64 bits]	I	Pins on which the new time value is presented.
value_out [32 bits]	O	Pins on which the higher or lower half of the time value is send to.
int_time	O	Timer interrupt. This line is raised if the interrupt counter reaches the indicated interrupt timer value.
second_half_ready	O	This will tell if the second half of the to be read time value is ready to be read. If this line is high, the second half will be read and this pin will be lowered.

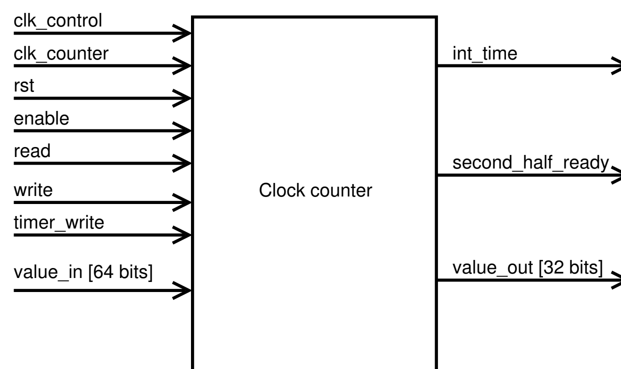

Figure 7: Input and output of the Clock counter component.

An overview of the input and output pins of the Clock counter can be found in Figure 7.

4.3.2 Timing diagrams

The timing diagrams of the waveform for the counter during normal operation are illustrated in Figure 8. The counter value takes one cycle to calculate which means that the counter value is updated the cycle after the counter clock cycle. The counter clock cycle does not have to be the same as the control clock as it may be counting the cycles for the partition counter. The waveforms for the interrupt show that an interrupt takes one cycle to be generated after the counter value is equal to the programmed timer value.

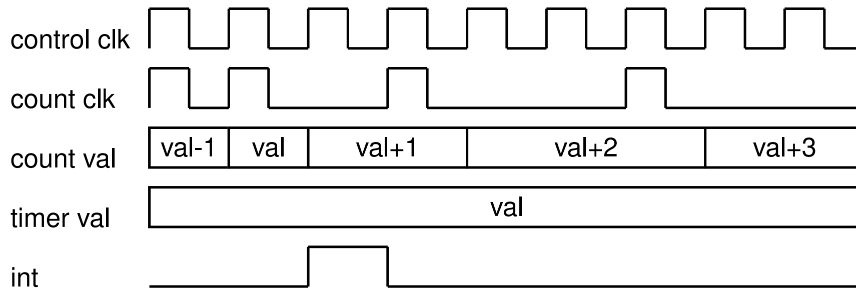

Figure 8: Timing diagrams of the Clock counter components.

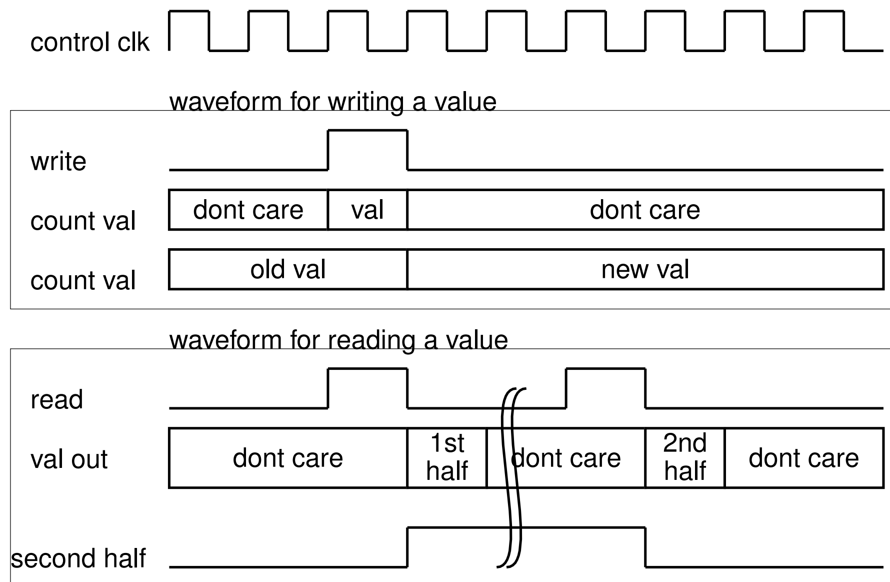

Figure 9: Timing diagrams of the Clock counter reading and writing operations.

The waveforms for the counter during reading and writing a value to the counter are illustrated in Figure 9. The waveforms show that writing a value takes one transaction whereas reading a value takes two transactions. One transaction needed for writing is required for making the counter respond as soon as possible to internal changes. Reading is only used for sending to the processor which

means it can send one half at a time since the counter value is twice the size of the FSL bus. The amount of cycles between the two reads does not have restrictions, but it does not allow a different read operation until the current read finishes. The counter stores the current value in a register when the operation is started to make sure no overflow occurs during reading.

4.4 Interrupt manager

The interrupt manager has to register all the interrupts which are generated either from one of the clock counters or from an external source. These interrupts are stored and then forwarded to a single interrupt line to the processor. Reducing all the interrupts to a single line is done since the processor has only one interrupt input. Since the interrupts are reduced and the processor needs to know what interrupt has been triggered the entire interrupt register can be read. If the appropriate routine for a certain interrupt is finished that interrupt can be acknowledged so the corresponding interrupt is lowered. To keep the interrupts for the appropriate programs a certain MASK can be set, so only specific interrupts will trigger the main interrupt. The interrupts which are non-responsive because of the mask are still registered, so when the appropriate program is resuming the interrupt is recognized.

4.4.1 Signal description

An overview of the input and output pins of the Interrupt manager is shown in Figure 10.

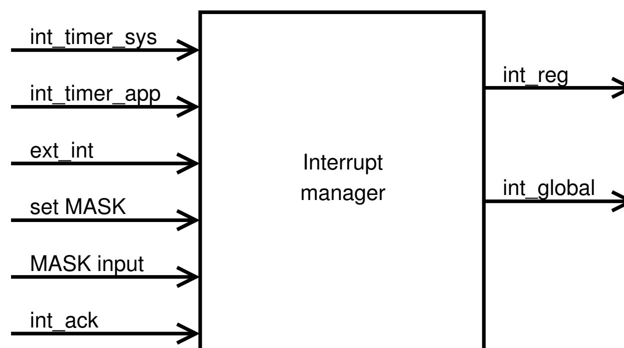

Figure 10: Input and output of the interrupt manager component.

The list of signals for the Interrupt manager can be found in Table 2.

Table 2: Signal listing for the Interrupt manager.

Signal name	I/O	Description
int_timer_sys	I	Interrupt time signal from the system counter.
int_timer_app	I	Interrupt time signal from the application counter.
ext_int	I	Interrupt inputs which are made externally from the ICTM module. This will trigger the int_global if the MASK forwards it.
set_MASK	I	If this pin is high write the current MASK value with the value from MASK input
MASK input	I	If set_MASK is high this value is read and used as the new MASK.
pending_int	I	If this signal is high the output interrupt register is ANDed with the MASk before the output.
int_ack	I	Acknowledge the interrupts where in this input are ones.
int_reg	O	These pins show which interrupt has been raised.
int_global	O	This pin is connected to the Microblaze processor and notifies if an interrupt is generated which is allowed by the MASK.

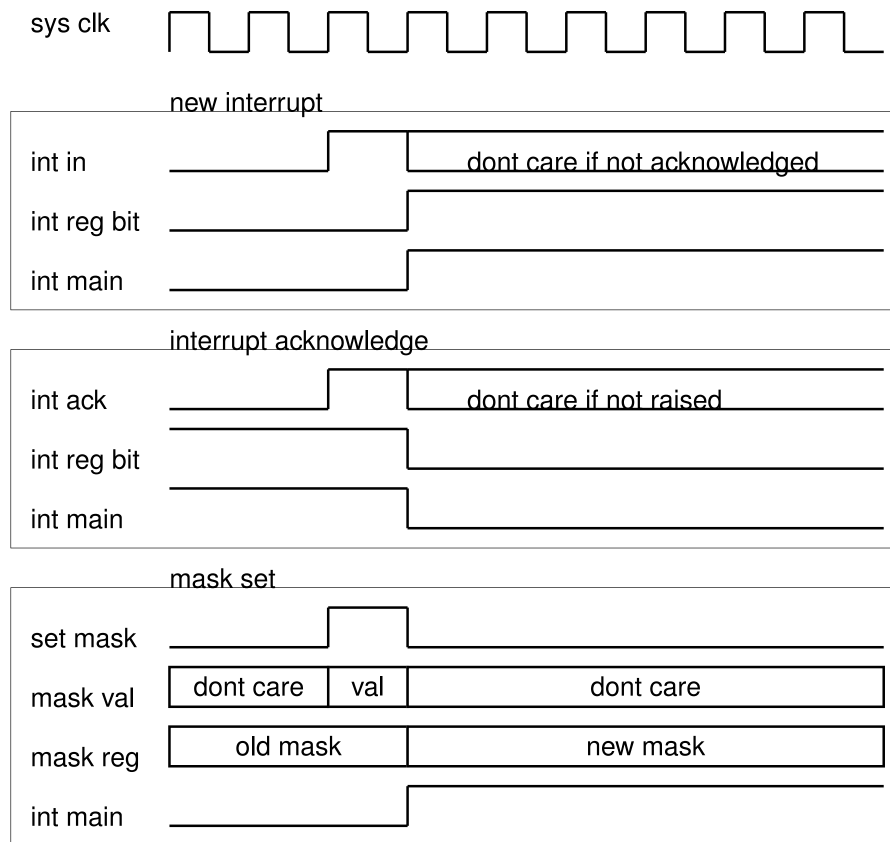

Figure 11: Timing diagrams of the interrupt manager component.

4.4.2 Timing diagrams

The timing diagrams of the interrupt timings can be found in Figure 11. They show the responds of the interrupt manager during generating and acknowledging interrupts. A new interrupt shows that it takes one cycle for the counter interrupt to register and forward to the processor. This is because of internal propagation. Interrupt acknowledge shows that the cycle after an interrupt is acknowledged the interrupt register is lowered. The mask set shows that when the mask is changed the interrupt signal is raised if the new interrupt is part of the new mask but not of the previous mask.

5 APIs of the ICTM

The ICTM is responsible for managing the interrupts, for which the necessary driver and APIs are presented.

5.1 Low level guidelines

Here the low level guidelines to design and implement the ICTM are listed, which if all are met should result in the behavior of the high level requirements.

- **OS slot: sets and acknowledges interrupts.** The OS should be able to set an interrupt in the future in system time space, and should be able to recognize and acknowledge the interrupt.
- **OS slot: length and period are exactly as programmed.** To be accurate on timings, the OS slot and the application slot should be exactly as long as programmed, generating a new system interrupt after exactly one period. The period of the OS slot is illustrated in Figure 4 (see Section 6). This includes preloading the pipeline for the application slot so the application slot takes exactly as long as programmed.
- **OS slot: counters are to be read out correctly.** The ICTM keeps several values including the timer values, previous interrupt times, and programmed interrupt times. These values should be able to be read from the ICTM in the correct manner, which should result in the correct value being received as is stored in the ICTM.
- **Application slot: context stored when interrupted and loaded correctly when scheduled.** The application should not be influenced by the system if it was interrupted. This means that all the registers should be saved on the stack, and the same values should be loaded from the stack when it resumes execution. In this way the application can continue with the exact same values as it had before it was interrupted.
- **Application slot: application should read the exact same application timer values.** The application should not notice a difference in the application counter if it is interrupted or not. This means that whenever the application is reading the application counter it should always give the exact same value independent of being interrupted or not. A special case includes reading the timer value within any critical region, and because of that possibly within the OS slot.

5.2 Interrupt management

A list of functions for the interrupt management are as the following:

- void **Int_ack(i)**: Acknowledge interrupt i.
- int_reg **Int_read**: Retrieve the interrupt register from the ICTM .

- int_reg **Int_pending_read**: Retrieve the interrupt register from the ICTM which is ANDed with the MASK.
- void **Set_mask(mask)**: Set the mask of the ICTM , regulating which interrupts should trigger.
- void **Int_set_sys_abs(TIME time)**: Raise interrupt at absolute time on the system counter.
- void **Int_set_sys_rel(TIME time)**: Raise interrupt in relative time on the system counter.
- void **Int_set_app_abs(TIME time)**: Raise interrupt at absolute time on the application counter.
- void **Int_set_app_rel(TIME time)**: Raise interrupt in relative time on the application counter.

5.3 Clock counters

A list of functions for the system counter are as the following:

- void **sys_cnt_set(TIME time)**: sets the system counter to time.
- TIME **sys_cnt_read**: read the current system time.
- void **sys_cnt_start**: start/continue counting.
- void **sys_cnt_stop**: stop counting.
- void **sys_cnt_init(TIME time)**: Sets the system counter to time and start counting.

A list of functions for the application counter are as the following:

- void **app_cnt_set(TIME time)**: sets the application counter to time.
- TIME **app_cnt_read**: read the current application time.
- void **app_cnt_start**: start/continue counting.
- void **app_cnt_stop**: stop counting.
- void **app_cnt_init(TIME time)**: Sets the application counter to time and start counting.

5.4 Advanced clock counters

A list of functions for the advanced system counter are as the following:

- void **sys_cnt_period_set(TIME time)**: sets the system period time.
- void **sys_cnt_period_enable**: enables the system counter period feature.
- void **sys_cnt_period_disable**: disable the system counter period feature.

A list of functions for the advanced application counter are as the following:

- void **app_cnt_period_set(TIME time)**: sets the application period time.
- void **app_cnt_period_enable**: enables the application counter period feature.
- void **app_cnt_period_disable**: disable the application counter period feature.

5.5 OS functions

A list of OS functions are as the following:

- TIME **last_int_time_sys**: retrieve at what time the last interrupt was asserted from the system counter.
- TIME **last_int_time_app**: retrieve at what time the last interrupt was asserted from the application counter.
- void **OS_slot_length(TIME time)**: Set how long the OS slot should be.
- void **OS_slot_bubble_set(char cycles)**: set how many cycles the bubble should be.

5.6 Operation Codes

The instructions for the ICTM must be transmitted over the FSL bus. The OP codes for all the different functions can be found in Table 3. The structure of the OPcode is that the first 8 bits are for selecting the component inside the ICTM , the next 8 bits are for choosing the function, and the remaining 16 bits are for a function argument (if required for the function). The structure can be found in Figure 12.

Figure 12: OPcode structure of the ICTM .

Table 3: Signal listing for the Interrupt manager

comp	func	function argument	send time?	selected function
Interrupt				
0x01	0x01	interrupt #	no	Int set
0x01	0x02	interrupt #	no	Int ack
0x01	0x03	none	no	Int read
0x01	0x04	none	no	Int pending read
0x01	0x05	MASK	no	Set MASK
0x01	0x06	none	yes	Int_set_sys_abs
0x01	0x07	none	yes	Int_set_sys_rel
0x01	0x08	none	yes	Int_set_app_abs
0x01	0x09	none	yes	Int_set_app_rel
System counter				
0x02	0x01	none	yes	sys_cnt_set
0x02	0x02	none	no	sys_cnt_read
0x02	0x03	none	no	sys_cnt_start
0x02	0x04	none	no	sys_cnt_stop
0x02	0x05	none	yes	sys_cnt_init
0x02	0x06	none	yes	sys_cnt_period_set
0x02	0x07	none	no	sys_cnt_period_enable
0x02	0x08	none	no	sys_cnt_period_disable
Application counter				
0x03	0x01	none	yes	app_cnt_set
0x03	0x02	none	no	app_cnt_read
0x03	0x03	none	no	app_cnt_start
0x03	0x04	none	no	app_cnt_stop
0x03	0x05	none	yes	app_cnt_init
0x03	0x06	none	yes	app_cnt_period_set
0x03	0x07	none	no	app_cnt_period_enable
0x03	0x08	none	no	app_cnt_period_disable
0x03	0x0b	none	no	app_cnt_read_fapp
OS functions				
0x05	0x03	none	no	read_interrupt_time_sys
0x05	0x04	none	no	read_interrupt_time_app
0x05	0x05	none	yes	OS_slot_length
0x05	0x06	none	no	read_error_state
0x05	0x07	length	no	write_app_preload_length
0x05	0x08	none	no	read_extra_work

5.7 Timings for the ICTM

The timing diagrams for the provided functions of the ICTM are illustrated in Figure 13. The waveforms show that the module reacts as soon as possible except if an internal propagation is required in order to meet timing constrains. These propagations are displayed in the waveforms. Because the FSL bus is 32 bits wide and writing the counter needs 64 bits, the transactions are divided in several parts. Sending or receiving values must be finished first before sending a new operation. During a reading transaction the signal que_full of the FSL bus must be raised in order for the processor to notify that a value is ready to be read. If this value is not raised the processor stalls and waits for this signal to raise. The last waveform shows that, when the period is enabled on a counter, that counter sets a new timer value if an interrupt is raised. The new timer value is equal to the previous value plus the programmed period value.

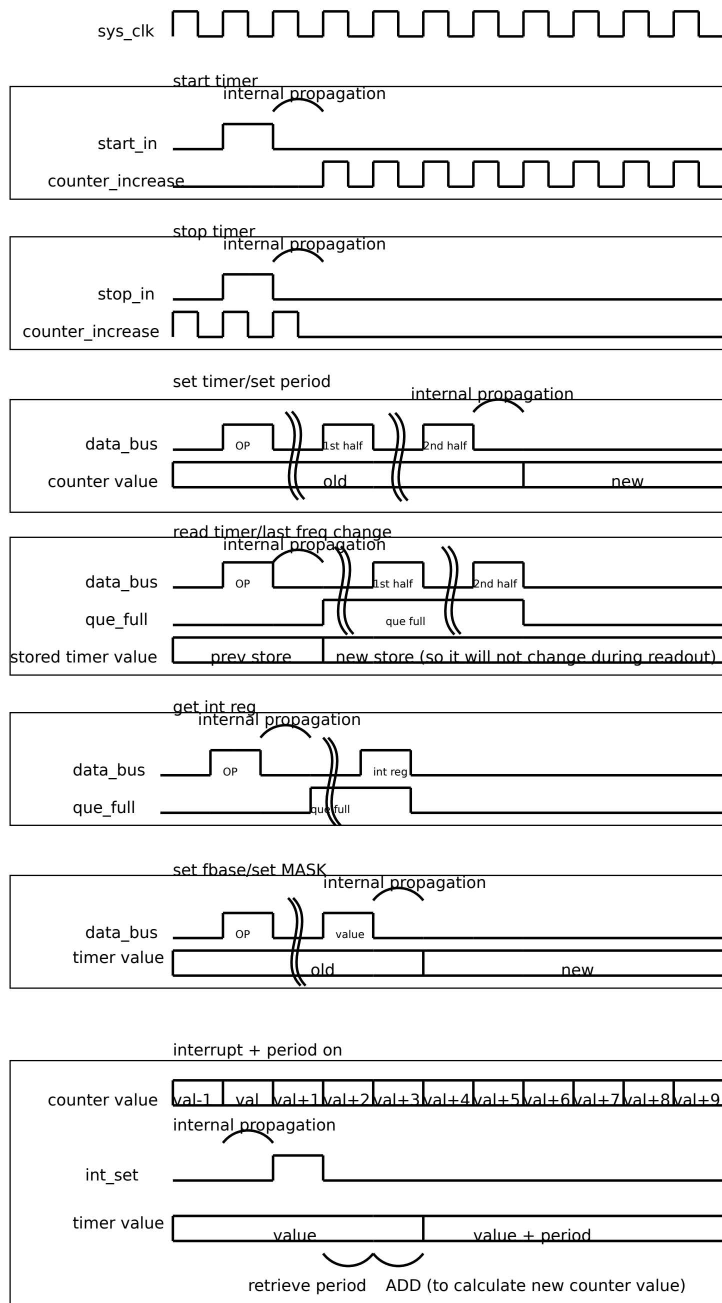

Figure 13: Timing diagrams of the ICTM functions.

6 Requirements

The ICTM module has been designed and implemented, in order to make sure that the partitioned applications cannot affect each other by even a single clock cycle [7]. We state nine general requirements on what the ICTM ultimately should achieve, as the requirements on software and hardware architectures [6]. The requirements are followed by a comment regarding the extent to which it is fulfilled by the research concept in this document.

1. The RTOS must support multiple concurrent applications, each consisting of a set of communicating tasks.

   It can be supported by the inter- and intra-application schedulers in RTOS (see Figure 1).

2. An application's tasks may be mapped on one or more processors.

   It is one of the assumptions of the T-CREST project and can be supported by the intra-application scheduler.

3. The RTOS must partition applications, in the sense that the worst-case and actual timing behaviours of any application are not affected by the absence or behaviour of any other application.

   It is supported by the time partitioning and interrupt virtualization mechanisms implemented in this document.

4. Each application must specify its own task scheduler, specific to its model of computation.

   It is supported by T-CREST platforms.

5. The following models of computation shall be supported: cyclo-static dataflow, Kahn process networks, time-triggered.

   Based on the interrupt virtualization techniques introduced in this document, cyclo-static dataflow and time-triggered models could be implemented and supported in OS schedulers on T-CREST platforms. Since Kahn process networks are not analyzable at design time, they are out the scope of the T-CREST research concerns.

6. A task scheduler may be non-preemptive (cooperative) or preemptive.

   Besides the support on cooperative scheduler, the developed device drivers and APIs provide time control on task level, which can be used by the task scheduler to implement preemptive scheduling.

7. A preemptive task scheduler shall be able to allocate a cycle budget to a task. The task will be preempted in the application when the cycle budget has been depleted.

   The intra-application scheduler can manage the cycle budget for tasks, based on the device drivers and APIs of the implemented ICTM module.

8. A preemptive task scheduler shall be able to allocate a deadline for a task. The task will be preempted at the latest time before the deadline when the application is active. The deadline shall be given as an absolute time.

   The intra-application scheduler can manage the deadline for tasks, based on the device drivers and APIs of the implemented ICTM module.

9. Each interrupt source, such as a hardware device, shall be assigned to and handled by one application. Interrupt arrival and handling shall not impede partitioning.

   The inter-application scheduler can exploit interrupt handling and virtualization mechanisms, based on the device drivers and APIs of the implemented ICTM module.

Here we list all requirements in aspect CORE and scope NEAR from Deliverable D 1.1 that are relevant for the processor work package. NON-CORE and FAR requirements are not listed here. The requirements are followed by a comment regarding the extent to which it is fulfilled by the research concept in this document.

P-0-505 Preemption:

The system shall provide means to implement preemption of running threads; these means shall enable an operating system or execution library to suspend a running thread immediately and make the CPU available to another thread.

The implemented interrupt structure has an interrupt management module to either forward external interrupts or generate timing dependent interrupts to preempt running threads.

P-0-506 Priority-preemptive scheduling:

The system shall provide means to implement CPU-local priority-preemptive scheduling without migration of threads between CPUs.

The implemented interrupt structure is independent of the scheduling algorithms adopted on processors. Therefore, priority-preemptive scheduling can be implemented.

P-0-508 Interrupts:

The CPU shall support interrupts.

Besides the original external interrupts and software interrupts, the developed device drivers and APIs can be used to keep track of the timing budget for applications and tasks, and trigger interrupts in scheduling.

P-4-014 Time control:

The processor shall support time control instructions. Such instructions are needed to implement interrupt virtualization.

The developed device drivers and APIs provide time control on both application and task levels, which can be used by the processor.

7 Conclusions

In this document, we have introduced how non-real-time tasks and real-time tasks can co-exist in predictable systems based composable preemptive schedulers. Virtualizing I/O interrupts, such that they are contained in the virtual platform belonging to the application itself, is therefore proposed to isolate the timing interfere amongst multiple applications. Consequently, how to design and implement the interrupt virtualization intellectual property (IP) core (i.e. the ICTM) is then presented. The device drivers and APIs for the ICTM are designed as well to provide the abstract programming interface on low level hardware-dependent functions.

Based on interrupt virtualization, the maximum execution time of each application and the time to serve the interrupt are bounded and known, so that the response times of other applications can be assured when critical actions are needed in a *partitioned system*. The device drivers and APIs are OS independent. They allow application specific schedulers to keep track of the timers and manages the interrupts in multi-application systems.

The work is to ensure real-time requirements of multiple applications when they share resources, and to prevent interference that may invalidate the timing requirements [6]. This work has been carried out as a part of the T-CREST project.

References

[1] INTEGRITY. http://www.ghs.com/products/rtos/integrity.html.

[2] LynxOS-178. http://www.lynuxworks.com/rtos/rtos-178.php.

[3] PikeOS. http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/.

[4] Avionics Application Software Standard Interface. *ARINC Specification 653*, January 1997.

[5] Felix Bruns, Shadi Traboulsi, David Szczesny, Elizabeth Gonzalez, Yang Xu, and Attila Bilgic. An evaluation of microkernel-based virtualization for embedded real-time systems. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 57–65, Washington, DC, USA, 2010. IEEE Computer Society.

[6] Eindhoven University of Technology and Technical University of Denmark. *T-CREST project. D 2.2 Concepts for Interrupt Virtualisation*, 2012.

[7] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–24, 2009.

[8] Gernot Heiser. The Motorola Evoke QA4A case study in mobile virtualization. 2009.

[9] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: space-time partitioning in a manycore client os. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[10] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. *Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach*, volume 18, pages 11–21. OASICS, 2011.

[11] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, March 2003.