CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Faculty of Computer Science
Real-Time Systems Group

Diploma thesis

“Porting eCos to the Analog Devices BLACKfin DSP”

André Liesk

Chemnitz, October 2nd, 2006

student : André Liesk, 30562
born on November 13, 1982 in Hoyerswerda
supervisor : Dr. Robert Baumgartl

Declaration of Authorship

I hereby declare that the whole of this document is my own work unless explicitly stated otherwise in the text.
This work is being submitted to Chemnitz University of Technology as a requirement for being admitted to the diploma of Computer Science (“Diplom Informatiker”).
I declare that this work has not been submitted in whole or in part for any other degree.

Table of Contents

Task

The task of this diploma thesis is “Porting eCos to the Analog Devices BLACKfin DSP”.

The open-source-based operating system eCos has gained considerable attraction within the real-time community. Its fine-grained customization mechanism, small memory footprint and openness make it desirable in many embedded projects. It has been ported to a broad range of micro controllers. So far, the Analog Devices BLACKfin DSP family is not among them. Therefore, the goal of this thesis is to port the hardware abstraction layer of eCos to that platform. A new architecture, platform and variant must be defined. This seems to be a fairly complex task.

It is intended to use that port for teaching purposes and system research within the real time group of Chemnitz University.

Abstract

This thesis covers the work to combine the two worlds of the hardware platform of the BLACKfin by Analog Devices and the software based on the eCos operating system to provide a foundation for embedded real-time applications to build on to benefit from the best aspects of both. This document will therefore outline the main objectives of this thesis followed by an overview of the functionality provided by eCos and the BLACKfin.
It will further outline the steps required to combine both by porting the hardware abstraction layer and device drivers for the BLACKfin architecture to eCos.
Prior to detailing selected implementations of particular code segments of special interest this thesis will outline the design and concept considerations involved and the conclusion drawn in order to provide a working HAL. After describing the current state of the hardware abstraction layer port conducted as part of this thesis this document will provide an evaluation of the implementation itself the benefits as well as possible limitations.
To provide a conclusion to the work outlined in this document further possible questions of interest for future work based on the results of this thesis will be provided.

1. Introduction

1.1. Motivation

In today's world the use of computing devices has become pervasive in nearly any part of the all day life. These devices have miniaturized and embedded in many devices of daily use beginning from the coffee machine to the safety critical airbag. As these embedded devices have different requirements to their environment it would be desirable to have an operating system being as flexible in its configuration as our world is versatile.
Due to memory limitations and real-time requirements of some equipment like airbag or audio devices the system should not just provide a small memory footprint but also be optimized with regard to time and speed. One operating system that has gained substantial support and is provided as open source is eCos. A hardware architecture that promises the to provide processing power and power management facilities as well as a vast variety of system devices is the BLACKfin by Analog Devices.
This thesis will try to combine both the hardware and software platforms to provide a foundation for real-time applications to build on.

1.2. Goal of the thesis

The task of this thesis is to port eCos to the Analog Devices BLACKfin family. In order to perform this complex task this thesis will perform the porting process based on the BF-533 STAMP board. This board features a Analog Devices BLACKfin, the BF-533 and is an application example for a design based on the BLACKfin processor. The board is currently supported by uboot a boot loader and uClinux a Linux variant for micro controllers.
The goal of the thesis is to port eCos in such a way that it will support the BLACKfin architecture and in particular the BF-533 STAMP processor. In addition the devices on the STAMP board should be supported if possible.
The first main aspect of the porting process is to provide a working and functional hardware abstraction layer that supports most or all of the functionality found in the existing HALs.
The second main objective is to provide a full support for RedBoot as boot manager to replace the existing uboot on the STAMP board.
The third main objective is to reach a support for the architecture that allows eCos and applications for eCos to be build with the default configuration and networking support. This includes support for the system devices on the STAMP board.

2. State of the Art

2.1. eCos

2.1.1. General idea of eCos

eCos is an embedded Configurable operating system designed for embedded devices and application. It is a portable system currently supporting more than 10 different architectures including ARM, MIPS, IA32 and Power PC. eCos as an operating system provides facilities for synchronization, scheduling, device drivers and compatibility layers for programs written for different operating systems.
It also provides network stacks and simple applications such as an HTTP server.
All parts of eCos can be configured based on the needs for the services required for the application in the embedded environment.
In addition eCos will provide a small memory footprint. The size of this footprint will vary depending on the configuration options that have been chosen as well as the program it will support. (ECOS 2006a-d)

2.1.2. Structure of eCos

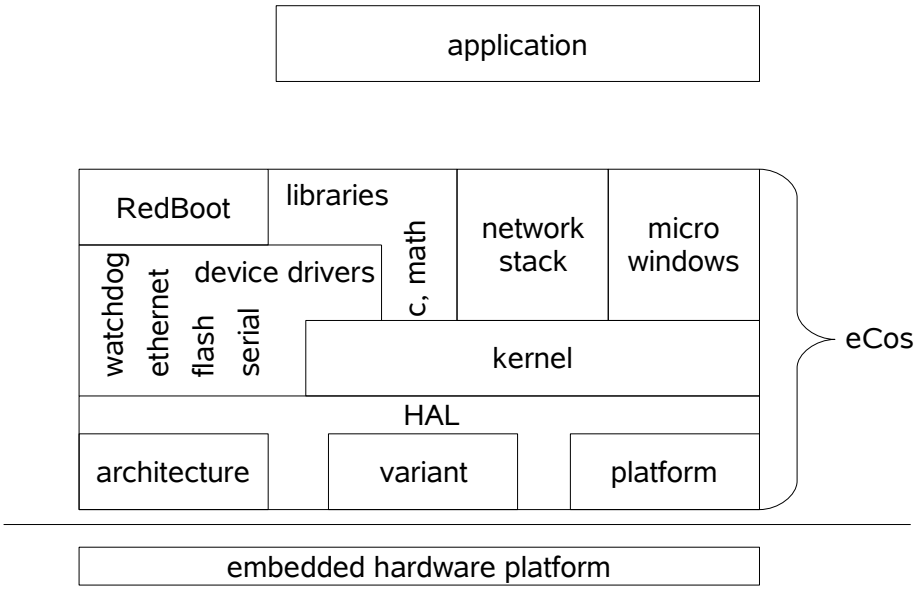


Illustration 1: simplified structure of eCos

Illustration 1 is showing a simplified view on the general structure of eCos. The view is simplified as not all possible layers are shown. Some layers have interdependencies that are also not shown here. The picture however gives a structural overview that is sufficient enough for the purpose of porting eCos to a different hardware platform.

As shown eCos is a layered operating system. The first layer above the hardware platform is called HAL or hardware abstraction layer. The task of this layer is to abstract from the underlying hardware and provide a common interface for all hardware platforms supported by eCos to the higher layers. This enables eCos to be easily ported as most parts of the operating system do not require changes. While most upper layer code is written in either C or C++ the HAL is written in assembler and C only.
Another layer that might be affected by a porting process are device drivers.
Each layer will provide services to upper layers while it uses the services of the layer below to support its own tasks. The application is at the top of this layered stack. It will ideally only use services provided by the layer directly below. This means that if an application does only use services provided by a subsystem of eCos it is portable to any architecture supported by eCos that allows these subsystems to run.
Picture 1 also implies dependencies between the layers. The application has to include all layers into its configuration of eCos it depends on. In addition all dependencies of these layers have to resolved.
The minimum requirement for an application to actually use eCos is the hardware abstraction layer. All layers above the HAL are optional and may be included as required by the application. This demonstrates the extent to which eCos can be freely configured and adapted to the embedded environment. (ECOS 2006a-d)

2.1.3. Application programmers view on eCos

For the application programmer eCos is not so much an operating system already running on the target platform with programs being loaded dynamically into the system.
While this is possible it does not represent the usual approach.
eCos will look to an application programmer as a library the program is statically linked against. This means that in order to link the program it is first necessary to configure and compile eCos itself for the specific needs of the application.
The actual program that will run on the target platform consists of the application code as well as the code for the operating system eCos. When the code gets executed on the target system eCos will boot and setup all the services configured prior to invoking the application. (ECOS 2006a-d)

2.1.4. Services provided by eCos

eCos provides many different services based on the configuration. In addition to this a service can be implemented in different ways and options.

Scheduler

One example service that is provided by the kernel is the scheduler. This scheduler has more than one possible implementation. The specific implementation used for the application program can be selected during the configuration process based on the requirements of the program. The scheduler itself is a preemptive, priority based scheduler. The two most important implementations of it are the bitmap and the multi-level queue scheduler. Both represent the same idea. Threads have priorities assigned to them. The thread with the highest priority that is runnable will be selected for execution by the scheduler. Both schedulers can be configured in the amount of different priority levels they support. (ECOS 2006a-d)

The key difference is the way they implement these priorities. While the bitmap scheduler has a simple bitmap where one bit represents a thread that is runnable the multi-level queue scheduler has a queue holding all threads that are runnable for a given priority level. The main difference from the application programmers point of view is that the bitmap implementation supports only one thread per priority level while the multi-level implementation supports more than one.
For systems that require multiple threads to run at the same priority it is impossible to use the bitmap scheduler. The multi-level queue scheduler also supports another feature. Due to the fact that more than one thread can share the same priority level it is necessary to decide which thread to select for execution.
Based on the configuration the scheduler will either apply cooperative scheduling among the threads of the same priority level or time slicing.
In the first case the threads have to yield the processing time to another thread of the same priority level by either nominated or anonymous yielding. The threads are subject to cooperative multitasking. However, a thread will always be subject to preemption if a higher priority thread becomes runnable.
In case of time slicing the threads will be preempted by the processor after consuming their time slice. The length of this period that the thread can run before getting preempted is a configuration option. (ECOS 2006a-d)

Synchronization

The eCos kernel also provides different synchronization mechanisms.
The first of them is the semaphore. Binary and counting versions of semaphores are supported by the kernel. In addition to semaphores the kernel also supports message boxes, flags and message queues, mutexes and condition variables. (ECOS 2006a-d)

Boot manager RedBoot

RedBoot as shown in picture 1 is not a layer of the operating itself as more an application program in its own right. The RedBoot boot manager only requires the HAL and some support subsystems to run. It does not use the kernel or higher layers.
The purpose of this application is to act as a ROM monitor. A ROM monitor is an application typically residing in a read-only memory that provides services like diagnostic output, debugging amongst others to programs running in RAM. Both, the application and the ROM monitor can either co-exist and share a single virtual vector table or the application might choose to replace the ROM monitors services. It might do so either on a per service basis or as a whole.
RedBoot as a boot manager also supports depending on its configuration FLASH devices and booting of ELF binaries. These binaries can as well as other data can be loaded via serial connection, from ROM or via ethernet. In order to provide the ethernet networking services RedBoot implements a simple network stack that allows images to be transferred either via TFTP or HTTP. (ECOS 2006a-d)

2.2. BLACKfin processor family

2.2.1. General notes

The BLACKfin is a digital signal processor family developed by Analog devices.
It consists of many representatives featuring different cache sizes, core speeds, on-chip devices and even dual core facilities.
The processor caters for audio and video signal processing with single instruction multiple data (SIMD) instructions. In addition the processor allows one 32 bit instruction and two 16 bit instructions to be packed into a 64 bit word for parallel processing.

2.2.2. The processor core

Illustration 2 shows the main features of the processing core of the BLACKfin architecture. The processor provides two main register sets divided in general data registers and pointer registers. It has 8 of each type while two of the pointer registers have special functions as stack and frame pointers. In addition the stack pointer register can refer to two different hardware registers depending on the mode of operation. The processor supports user and supervisor mode while the stack pointer register of user and supervisor mode do access different hardware registers. This means that the BLACKfin will automatically perform a stack switch to the supervisor stack when changing to supervisory mode. In order to provide access to the users thread context the BLACKfin also provides a user stack pointer register not shown in the picture. In addition to these general registers the core also provide special index register sets. This set of 4 sets with 4 registers each shown in light blue allow indexing of data. They support circular buffering if required.

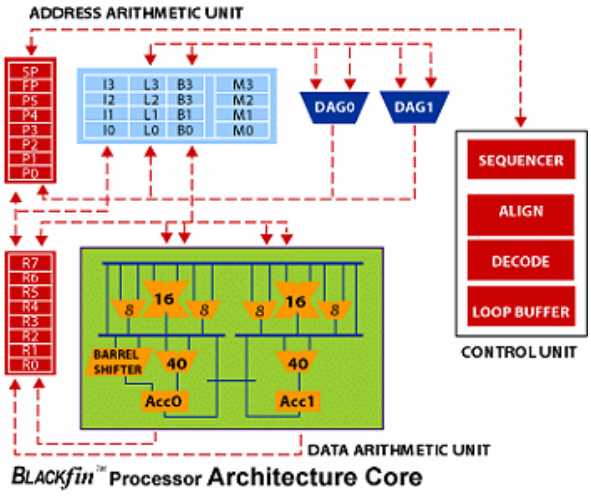


Illustration 2: BLACKfin architecture (ANALOG 2006c)

The main processing part of the core is done by two 16 bit multiply accumulators (MAC) and two 40 bit arithmetic logical units (ALU). In addition to these functional elements the core also provides four 8bit ALUs and a single barrel shifter.
The processor uses a RISC like instruction set with varying instruction length of either 16 or 32 bit. This allows frequent instructions to be encoded shorter than instructions that are less frequent or contain long immediate values.
Each of the registers can be accessed as one 32bit register or two 16 bit registers.
In addition to the features already mentioned the BLACKfin also supports two hardware zero overhead loops that can be nested to cater for two-dimensional hardware zero overhead loops. The processor also supports instruction flow control mechanisms like jumps, subroutine calls as well as conditional jumps and conditional move instructions. (ANALOG 2006a, 2006b)

2.2.3. On-chip devices

The BLACKfin processor family features a variety of different on chip devices that can be used to access data external to the core.

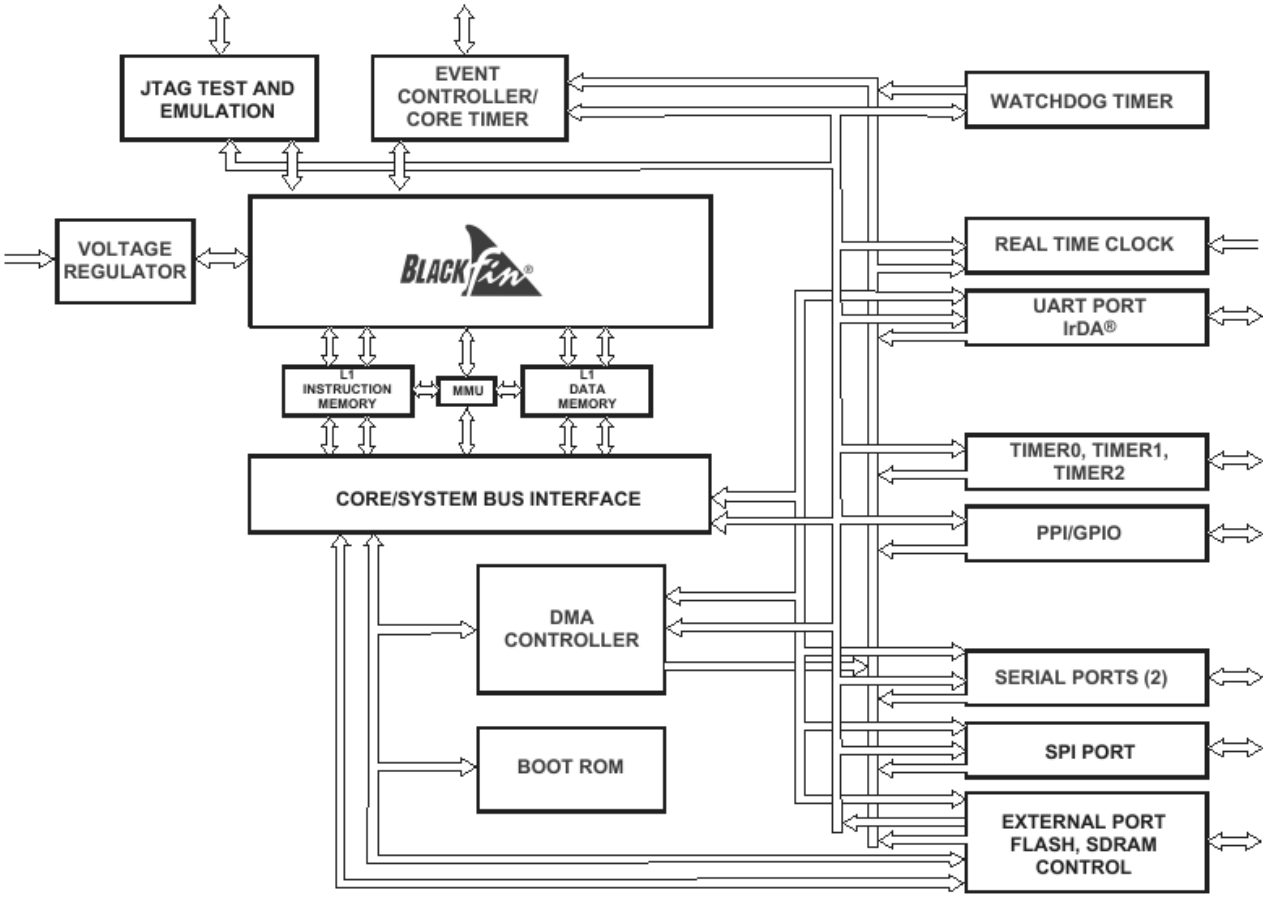


Illustration 3: BLACKfin devices of the BF-533 (ANALOG 2006d)

Illustration 3 shows an overview of the devices supported by one member of the BLACKfin processor family, the BF-533. This specific member has been chosen as it is the representative used for this diploma thesis. As shown in the picture the processor has a built-in direct memory access (DMA) controller. This controller supports multiple channels between external devices and memory. The BLACKfin also provides a built-in memory management unit (MMU). This unit however does not support address translation or virtual memory in general as the MMU of the Intel 386 does for instance. The purpose of the MMU on the BLACKfin is to support caching and page protection. The MMU therefore provides a page approach that allows pages of different sizes to have different properties. A page can be protected from reads and writes in user but also in supervisor mode. In addition the caching strategy can be defined on a per page basis. The memory system of the BLACKfin can access a maximum of 128 MB of synchronous memory in addition to a maximum of 4 MB of asynchronous memory. While some processor variants support a 32 bit bus to the memory most variants including the BF-533 only provide a 16 bit wide interface to the main memory. (ANALOG 2006a, 2006b)

The picture also shows on the left side the voltage regulator that allows a dynamic power management to be applied to the core. It is possible to change the core and system clock speeds independently.
In addition to these on chip devices necessary to operate the processor core the BLACKfin also provides additional system devices.
Among those extra devices is a watchdog that allows the system being reset, or an interrupt being raised in case it is not reset in the specified time.
The BLACKfin also provides a real-time clock that allows the days, hours, minutes and seconds since a specific date to be counted. In addition this real-time clock also supports second, minute, hour alarms as well as alarms on a specific time of the day.
Furthermore, the clock the processor also features four timers of which one is a dedicated core timer that will directly enter the core while the remaining three are general purpose timers.
A UART controller that is PC compatible and also supports IRDA is available too.
Additional devices that can be accessed are the parallel peripheral interface (PPI), two serial port controllers (SPORT), the serial peripheral interface (SPI) and the general purpose input/outputs or programmable flags (GPIO).
Some other variants like the BF-537 also feature a controller area network (CAN) device as well as an ethernet MAC device on core. The amount of other devices like timers and UARTs can very depending on the family member. (ANALOG 2006a, 2006b)

2.2.4. Event system

Illustration 3 shows the event controller that is responsible for dispatching events that enter the core. This event controller is a priority driven controller with support for 16 different core priority levels. It allows the assignment of entry points of service routines for each of the vectors to be programmed in a event vector table (EVT) which is accessible as a core MMR. The event controller also allows the 11 lower priority events of the 16 events inputs to be masked while the higher 5 events cannot be masked. These non-maskable events include reset, non-maskable interrupt and exceptions. The highest priority event is the emulation event that will be triggered either by a special instruction to enter emulation or via the JTAG interface supported by the processor for hardware debugging.
(ANALOG 2006a, 2006b)

3. Porting eCos tasks

In order to port eCos to a new architecture it is necessary to first identify the individual parts and subsystems of eCos that have to be ported. Based on this process the existing interfaces have to be determined as they must be kept during the porting task.
Due to the structure of eCos most parts and subsystems are not architecture dependent. Amongst those subsystems that do not have to be ported are the kernel and upper layers. The part of eCos most affected by a port is the hardware abstraction layer.

3.1. Porting the HAL

The hardware abstraction layer is an integral part of every eCos configuration.
A valid eCos configuration must select a HAL to be used as all subsystems will build be upon it. The HAL itself however is subdivided into three different parts itself.

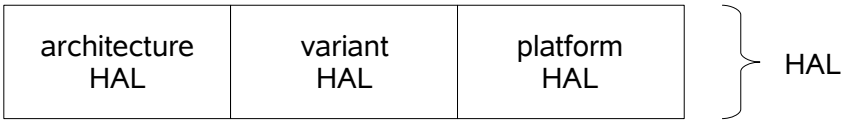


Illustration 4: general structure of the HAL

The HAL consists of the architecture, the variant and the platform HAL as shown in picture 4.
The architecture HAL represents all functionality of the processor family that are common amongst all members. The variant HAL however represents the a specific member or group of members of the family with similar attributes. This part of the HAL will most likely represent things like caching and interrupt mappings as these might change for different members of a family. One example are different cache sizes and a different amount of interrupt sources. With the general handling of interrupts being universal for all members it might be placed in the architecture HAL.
The platform HAL contains all parts that belong to a particular platform. This means everything that is not on the processor itself but on the board the processor is mounted on will most likely be placed here. The support of board specific devices in the platform HAL is limited to necessary devices for startup and running the system as well as devices used for debugging. All other devices that are not required for a general simple working system will be implemented as drivers rather than in the HAL directly.
The platform HAL depends on a variant HAL which itself depends on an architecture HAL. The BLACKfin is a new architecture that is not currently supported by eCos. This means that the processor family will constitute a new architecture HAL. In order to support the STAMP board used for this thesis it will be necessary to create a variant HAL for the particular member of the processor family, the BF-533. In addition a platform HAL has to be created to support the STAMP board itself.
This is the most complex porting process when porting eCos. As the architecture is not currently supported all parts of the HAL must be ported.

3.2. Porting device drivers

Another tasks when porting eCos that is likely to occur with any kind of port are new device drivers.
The amount of work attached to porting device drivers will vary based on the differences between the architectures and if a particular device is already supported.
In case of the BLACKfin processor BF-533 and the STAMP board the porting process will include a driver for a wallclock device. This device will allow the kernel to tell the current time. In addition a device for a watchdog must be ported in order to support the watchdog of the BLACKfin processor.
The UART serial and network driver must be ported to support a serial and ethernet connection. The port will be based on the stable branch of eCos that is currently available.

3.3. The STAMP board

The STAMP board is project that is driven by http://blackfin.uclinux.org/projects/stamp.
The version used for this thesis is the ADDS-BF533-STAMP. This is the first version of this type of board that was released. At the moment there is also a newer version of the board featuring the BF-537 processor available. The thesis will evaluate the amount of work required to fully support the new platform with the BF-537 processor.
The main focus of this thesis will be the BF533-STAMP version.
The STAMP board with the 533 BLACKfin does have a variant that supports up to 500 MHz core speed. It is driven by a 11.0592 MHz crystal clock input. In addition the board has a SMSC 93c111 ethernet MAC and corresponding PHY chip that will be supported for ethernet networking.
The board also provides 128 MB of RAM and 4 MB FLASH memory as asynchronous memory. The BLACKfin BF-533 provides a PC style UART controller, SPI, PPI, programmable flags and SPORT as well as DMA.
Due to porting eCos to the BLACKfin architecture the main goal of this thesis will be to provide a working code base that supports all necessary devices and allows programs being written for eCos using these devices. The stable version of eCos does currently neither support PPI, nor SPI or SPORT. This means that due to missing driver classes no support for these interfaces will be included in this thesis.

3.4. Porting subtasks

The porting process to achieve the main objectives is still too complex. In order to be able to achieve the objectives it will be necessary to define subtasks that need to be completed. The combination of completed subtasks will yield the support of the main objectives lined out in the previous section.
The first task in order to start the porting process is to create a new architecture, variant and platform HAL based on an existing HAL. The porting guide (eCos 2006e) of eCos suggests to take the MIPS HAL as an example to start coding.
All MIPS code that is not portable to the new BLACKfin HAL will be removed. All functionality that has to be provided by the HAL in form of functions and macros will be empty. Based on this skeleton the next task will be to add code for specific parts of the HAL.

These subtasks within the HAL will be the context switching macro including the definition of the context. Following this a simple serial driver for diagnostic output will be provided in order to start using the most basic configuration of RedBoot.
In order to assist the porting process RedBoot will first be run as an application program loaded by uboot. Following the context switch macros and functions it will be necessary to support an interrupt and exception system in order to allow events of the core to be serviced. The interrupt and event system is not essential for RedBoot but mandatory in order to continue to the next step of supporting the eCos kernel.
The porting process will be assisted by the eCos provided test programs that will allow testing of specific parts and functions of the code.
By this stage context switching, exception and interrupt handling, the basic startup code as well as the input output macros will be functional.
To be able to use the newly written HAL it is necessary to create the configuration files and to enter the new parts of eCos into the database.
Beginning from now the hardware drivers for the UART and ethernet will be ported.
This allows data to be downloaded in RedBoot via ethernet instead of the serial connection. This will yield a faster development cycle as the download time via ethernet is only a couple of seconds compared to many minutes via the serial line.
From now on the remaining skeleton parts of the HAL that have not been essential so far will be programmed and tested.
In addition critical code of context switching, interrupt and exception handling will be checked specifically for correctness. This include checking for the size of operands, the order of instructions as well as the registers and memory locations used.
Particularly important is the fact that all registered that will be manipulated must have been saved prior to overwriting their values in order to allow the threads to continue their run without any changes of underlying data.
After all these features have been added and tested the main focus will shift to evaluating the result of the porting process. This will include running tests to establish the correctness of the code but also timing tests in order to determine the time critical variables like time for the core timer interrupt or creation of threads.
Eventually, the setup code for the platform will be added and RedBoot as the boot manager will be moved from RAM to the FLASH ROM memory and replace uboot as boot manager. With this the first application based on eCos has been successfully ported and proves the basic function of the newly ported HAL.

4. Design and Concept

4.1. Context switching

Context switching is one of the most crucial tasks of an operating system supporting multiple threads to run has to fulfill. The code has to satisfy a couple of conditions to be able to allow proper operation.
The most important condition is that all registers that have been identified as the context must be saved and restored. Due to the fact that context switching can occur quite often in a system, a couple of hundred times per second, the code to perform this task should be optimized with regard to time. It should also be considered whether the code is interruptible or not.

4.1.1. Types of context switches

Context switches in a system are usually done due to scheduling activities in the systems scheduler. In case of eCos this can either be the scheduler of the kernel or any other part that acts as a scheduler. The basic context switching is a part of the HAL's functionality and is independent from the particular use. The code however has to ensure that certain conditions regarding the processor state are always met.
Due to these facts the use of the context switching macros is prohibited in any priority level above the interrupt levels.
This means that context switching cannot be done while in any core priority level smaller than 5. This also means that it is not possible to invoke a context switch directly within an exception handler being processed with priority three.
eCos provides the abstraction of VSR's in order to hook into system events provided by the HAL. Due to the system's state requirements the only VSR that always allows context switching activities is the interrupt VSR. Any VSR servicing an exception, non-maskable interrupt or reset event on the corresponding priority level must not invoke the context switch macros.
In case of the use of the kernel, context switching is usually a result of a system call. This will either be the interrupt_end function at the end of the interrupt VSR or any system call that will may suspend the current thread.

4.1.2. System states

The context switching macro has to differentiate between the system states of eCos to ensure proper operation.
eCos assumes that it will always run in supervisor mode of the processor and that it always has all rights attached to this mode. The paradigm used by eCos to invoke system services is by function calls and not as common in most other operating systems by some kind of interrupt or exception event. This means that any thread invoking a system call must run at the appropriate processor privilege level in order to perform the task or it will fail to do so. The system state definition is one applied by the BLACKfin HAL onto the system and must be adhered by all parts.
The HAL differentiates four states. The first state is BFIN non-maskable state.
This level include all core priorities below 5. This means the set of emulation event, reset event, non-maskable interrupt, exception event and the reserved event 4.
As described in the previous section these events must not invoke context switching.

This is not a restriction to the system as context switching can still be invoked. In order to achieve this task the service routine can switch to a lower priority event that is not a member of the BFIN non-maskable state set and invoke the context switch from there.
The second group of events is BFIN system state. This state is achieved by any event with core priority smaller than 15. So any event at core priority levels 5 to 14 will be classified as a BFIN system state. The BFIN user state is core priority level 15.
The fourth and last state is BFIN forbidden state. This state applies to the processor being in user mode.

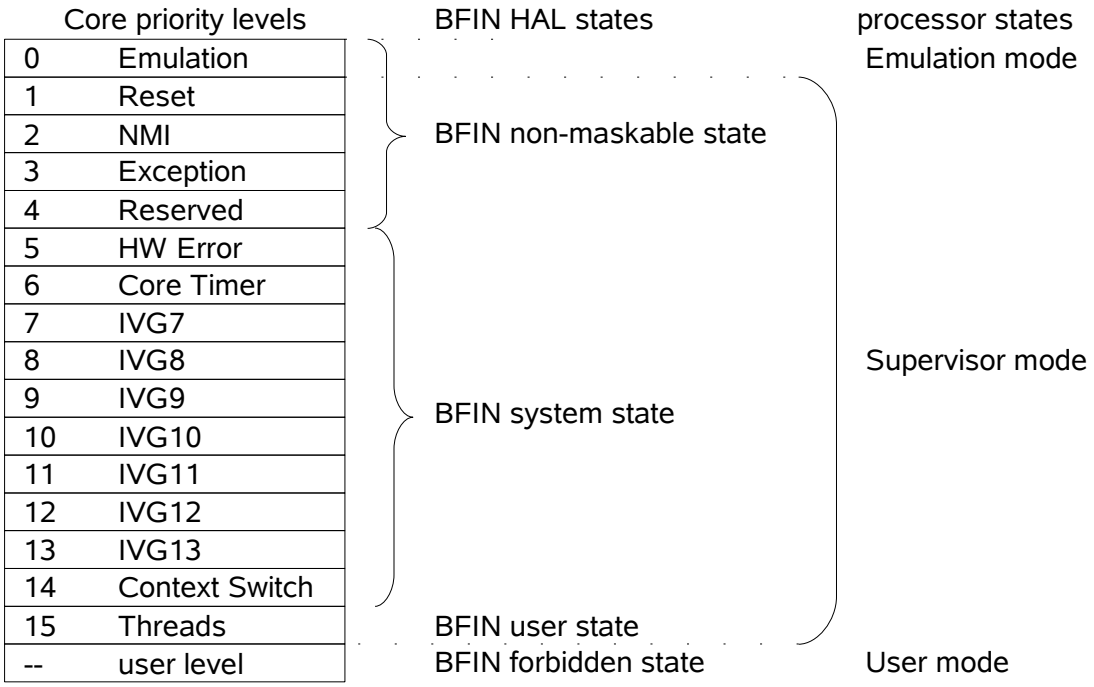


Illustration 5: priority state mappings

The difference between the two main states that are allowed to invoke the context switch macros is that no RTI is allowed while processing in the BFIN user state. This RTI would result in interrupt number 15 being finished and the processor would switch to user mode. eCos however will not work in user mode and it is the HAL's responsibility to ensure that the processor does not enter this mode.
Programs executed in eCos must only use subroutine call functionality unless they are exception or interrupt VSRs. This means that any thread will only use RTS to return from a subroutine function. As RTS does not affect the processors state it is save to be used in any state. Due to the structure of the interrupt VSR special care must be taken to keep the processor in the appropriate state. The context switching macro must store information necessary for the HAL to determine the state of the processor while the context got suspended. There is a single register that allows to determine the state by querying the bits set. This register IPEND records any pending interrupts. As long as at least one of these bits is set the processor is not in user mode. The IPEND register must be included in the threads context.
To continue the discussion of context switching it is necessary to define compatible core priority levels. Core priority levels are compatible if they belong to the same BFIN HAL state.

Direct switching without any changes to the state of the processor is only possible if the priority level of the thread to be suspended and the thread to be resumed are compatible. If the states are not compatible the processor state has to be adjusted prior to restoring the new context.
In order to demonstrate the fatal behavior of not adjusting the processor state in a switch between two incompatible state two examples have been chosen.

Case 1:
Assume the suspended thread got suspended at the end of an interrupt VSR by the interrupt_end call. This situation is likely to happen at the end of the core timer interrupt. The higher priority thread that will be resumed had been suspended by a system call to wait for an alarm while running as interrupt 15. As the timer interrupt will determine the alarm time of the thread it will cause a context switch. If the context switch would only restore the register set the resumed thread would continue running. It will be resumed within its system call that will finish with a RTS and return to the threads code. Because the thread has not issued a RTI instruction the timer interrupt has not been left. The thread will run with the core priority of the timer interrupt. This will prevent any interrupt with lower priority and the timer interrupt itself from being serviced meaning that the thread is currently not subject to preemptive scheduling. It is only subject to cooperative scheduling by invoking a function that will suspend itself.
This situation shows that the system will be in an undesired and faulty state.
The system might recover from this situation but also may not. It depends on the action of the thread whether and when the system will recover.

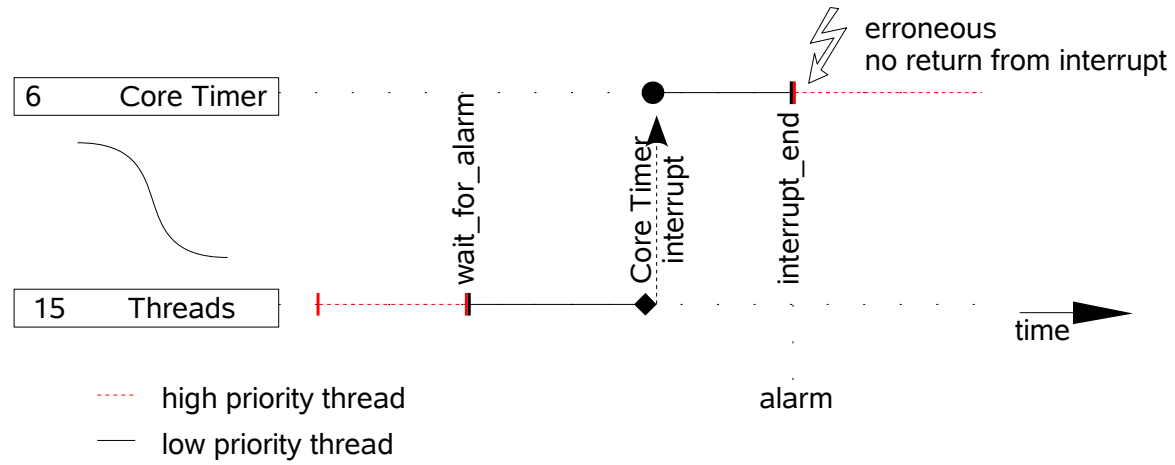


Illustration 6: case 1, no return from interrupt

Case 2:
Assume a lower priority thread that was suspended by an interrupt_end call in an interrupt VSR will be resumed by the higher priority thread calling a thread_suspend() system call on interrupt level 15. It is strictly forbidden to issue a RTI instruction while running at interrupt level 15.
Without state adjustments the context switch will restore the lower priority thread within the system call of interrupt_end. This call will return to the interrupt VSR. Now this interrupt VSR will restore its working registers before executing an RTI instruction to return from the serviced interrupt. But due to running on interrupt level 15 this RTI will cause the processor to clear the last bit set in the IPEND register and fall back to user mode. It is impossible for the system to fully recover from this state unless an interrupt 15 is raised by software. This is by convention of the BLACKfin HAL an illegal action. While interrupts will get serviced appropriately system threads will fall back to user mode or cause behavior similar to case 1.

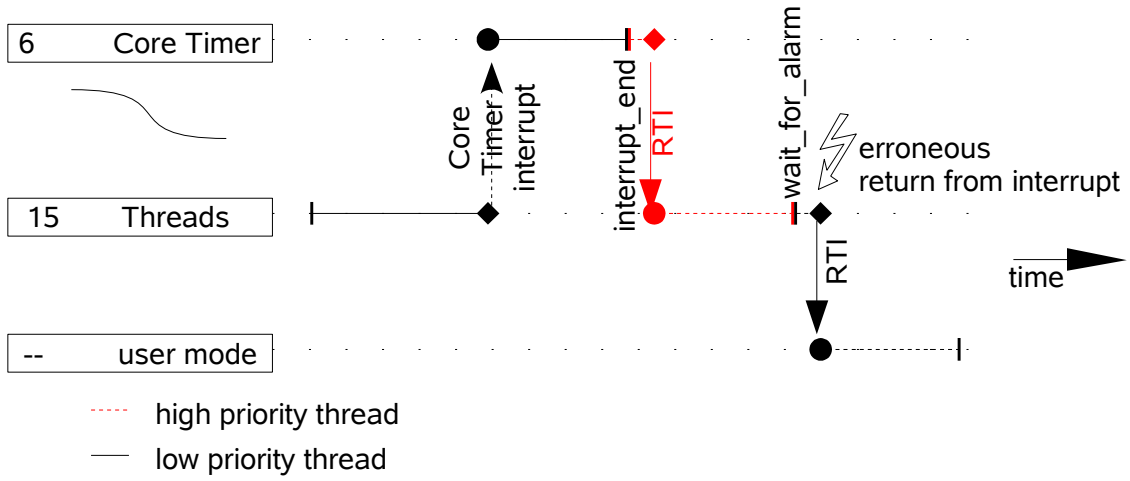


Illustration 7: case 2, erroneous return from interrupt

The two cases illustrate that it is necessary for the HAL to adjust incompatible system states prior to restoring the threads context.
The solution to the problem involves the following steps.
The HAL first has to compare the current state and determine whether this is compatible with the state of the thread that should be resumed.
If both states are compatible the HAL simply has to restore the threads context without further adjustments. If they are not compatible the HAL must decide based on the current or new state which action should be taken.

Case 1:
Switching from HAL system state to HAL user state requires an easy adjustment.
The HAL will set an appropriate value in the RETI register to point to the first instruction of the normal context restore function. After setting the RETI register the HAL just executes the RTI instruction. This instruction will return from the interrupt and return to interrupt level 15.
In order to achieve this function the HAL imposes a paradigm used by the eCos scheduler for all activities that invoke the macro. The macro must only be used within the last interrupt pending. If interrupts are nested only the last interrupt, the first that got serviced must execute the context restore macro.

Case 2, requires a bit more work than case 1. In order to switch from HAL user state to HAL system state the HAL must generate an interrupt. Interrupt 14 is reserved is reserved for this use by the HAL for internal use. The HAL will setup the vector to point to a special location within the context switch code. It will then disallow all interrupts except for interrupt 14 and cause interrupt 14 in software. Once this interrupt will be serviced the HAL will continue the execution within the context switch code. Immediately after switching to interrupt 14 the HAL will reallow interrupts to occur and restore the threads state.

4.1.3. Context

The BLACKfin has a large set of registers that must be considered to be included into the context of a thread.
I have identified the following register groups to be mandatory.
The first group covers the general purpose registers R0 to R7. These registers can hold any kind of value and can be used as operands to many instructions. As context switching cannot be compared to a normal function call it is necessary to save the entire set.
Also included in this group are the accumulator registers A0 and A1 and the register ASTAT. While the registers A0 and A1 are changed explicitly the ASTAT register is also subject to implicit changes. Any operation handled by either accumulator of the BLACKfin can as a result change the ASTAT register.
Furthermore, the pointer registers P0 to P5 as well as the frame pointer FP are registers that need to be saved.
The stack pointer SP also belonging to the pointer registers is handled separately. As the stack pointer points to the top of the context saved on the stack itself it is also used to reference the location of the threads context. In addition any direct restore of the stack pointer via a pop operation is impossible due to instruction set limitations.
The stack pointer needs to be saved as reference to the threads context at a location provided to the context switch code.
The third group are the return registers. The BLACKfin processor does not push the return address on the stack when performing a CALL or invoking an interrupt or exception.
It does save the return address in a special register depending of the type of event.
There are five different types which are mentioned in descending order: emulation, non-maskable interrupt, exception, interrupt, subroutine.
Each of these events has its own register to hold the return address. These registers, RETE, RETN, RETX, RETI and RETS must be included into the context.
The registers RETE and RETN should not be of that much importance as a context switch cannot be initiated by these events directly but they are included to cater for specific implementations of these VSR's.

The RETX register is of importance for the exception handling routine or for debugging purposes. For completeness, all five registers are stored.
Another set of registers that need to be included are the hardware loop support registers LCx, LTx, LBx that refer to the remaining iterations, top and bottom of the hardware loop.
Another group of registers are the index registers. This group of 4 sets with 4 registers each is used to index into memory. As each program can use its own set of these registers the values must be included in the context.
In addition to the registers directly used by the program registers describing the status of the processor and system at the time the thread is suspended must be included too.
These registers are:
The SYSCFG and CYCLES, CYCLES2 registers are important to restore the system state of the thread, for instance single stepping and cycle counting.
The SEQSTAT register holds the exception cause and other information. This information is of importance for exception handlers.
The IPEND register is included in the context as it is vital for restoring the context of the thread correctly.
The following registers are saved to support exception handling routines and debugging functions: ICPLD_FAULT, DCPLD_FAULT and USP and IMASK.

	0x0	**0x4**	**0x8**	**0xC**
SP+0x0	*ICPLD_FAULT*	*DCPLD_FAULT*	AW0	AW1
SP+0x10	AX0	AX1	*SEQSTAT*	*IPEND*
SP+0x20	*IMASK*	P5	P4	P3
SP+0x30	P2	P1	R7	R6
SP+0x40	R5	R4	R3	R2
SP+0x50	R1	R0	SYSCFG	CYCLES
SP+0x60	CYCLES2	I0	I1	I2
SP+0x70	I3	L0	L1	L2
SP+0x80	L3	B0	B1	B2
SP+0x90	B3	M0	M1	M2
SP+0xA0	M3	LT0	LT1	LB0
SP+0xB0	LB1	LC0	LC1	RETS
SP+0xC0	RETI	RETX	RETN	RETE
SP+0xD0	*PC as RETX*	ASTAT	USP	FP

Table 1: context of the BLACKfin

4.2. Exception handling

Exceptions are events that occur synchronous to the execution of the thread by the processor. While these events are usually generated by hardware it is also possible to generate 16 different exceptions in software on the BLACKfin processor.
This chapter will cover the exception handling by the processor in hardware and the implications for the real time characteristics of this approach. It will further classify the types of exceptions and determine which of these must be handled by the HAL and which can be handled by higher layers.
Based on these information I will describe two different strategies for handling exceptions and select the one most applicable.

4.2.1. Event flow of an exception

The BLACKfin processor family can differentiate a maximum of 64 different exception causes. This already includes the 16 software exceptions.
An exception can be either of two types. In case of a service exception the instruction causing the exception will pass the execution stage and commit. In case of an error exception however the instruction will not commit prior to entering the exception vector.
The processor will now start serving the exception event if no higher priority event is currently serviced. This means that exception should be avoided in the reset or non-maskable interrupt vector as well as in the exception vector itself.
The event will be serviced by the processor before any further instruction will be executed. The processor as shown in illustration 8 first switches to the exception priority level and starts executing the first instruction of the exception handler. The entry point to this handler is registered in the event vector table of the processor like all other core priority handlers are.
The exception handler will now executed and handle the exception. This might be done by simply ignoring the error in case of service type exceptions or by eliminating the error condition that has caused the exception. Once the handler finishes this task it will return to normal execution of the code and the core will leave the exception core priority level.

4.2.2. Side effects and latencies

While the processor is handling an exception all conditions for core priority handling apply. This means that no event with lower core priority can be serviced by the processor while the exception is handled. Illustration 8 shows one case in which this behavior might not be desired. As it can be seen an asynchronous interrupt event occurs while the processor is serving the exception. Due to the conditions just laid out it is not possible for the interrupt to get serviced immediately it will be delayed instead until the exception handler is finished. This is a problem in real time application as this will increase the latency for serving interrupts and make it impossible to use the system in real time environments.
Another downside of this approach is the fact that it is difficult to measure an average latency as well as a maximum latency for interrupts. Due to the exception handler the maximum latency is potentially infinite. It is possible to register any type of handler for an exception and eCos will also allow user threads to register their own handlers.

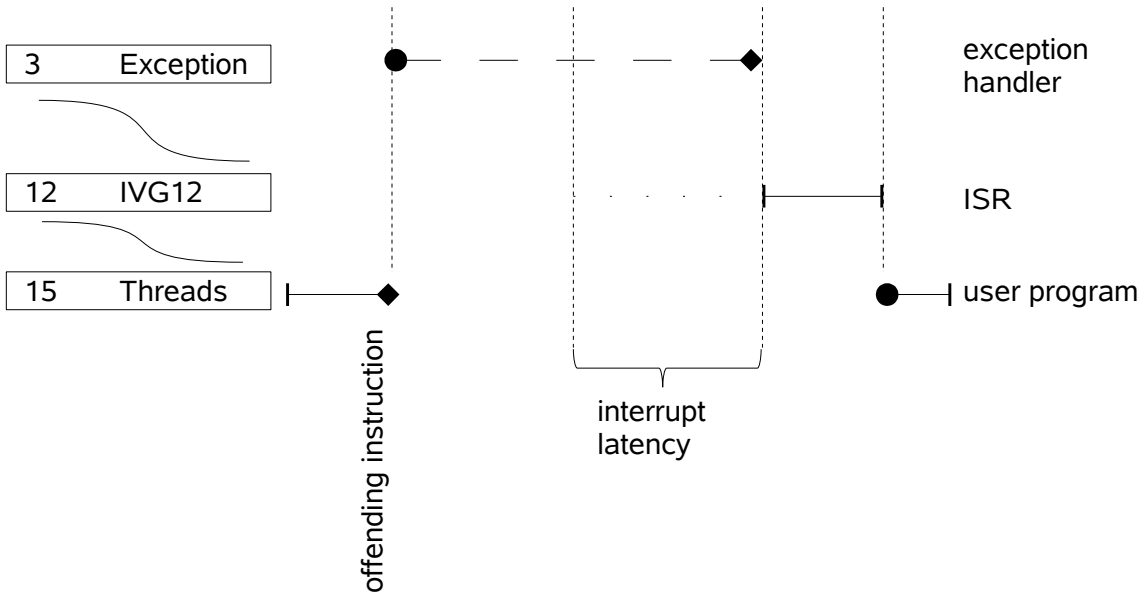


Illustration 8: exception handling

This is a potential problem. At first it is not feasible for the HAL to give best, worst case or average estimations if it has no control nor knowledge about the exception handlers of user threads. It is not difficult to imagine that a user thread is registering a handler for one of the software exceptions. This handler might be lengthy to discriminate between a lot of different options. But the handler could also be deliberately programmed to use up computation time. Due to the execution of the handler on core priority level 3 all other core priorities above are virtually disabled. This means that any exception acts also as a temporary disable for the entire interrupt system including the core timer interrupt on priority level 6. This will also mean that the thread causing the exception is for the time of the exception handler, not subject to scheduling. Even more serious though is the fact that all parts of the system that rely on the core timer interrupt will not work. These subsystems include counters, alarms and timers. Threads waiting to be resumed at a particular time will be delayed by as many core timer intervals as are missed due to the exception handler. This system behavior can have serious impacts on environments with soft and hard real time requirements. A system working like that is not able to satisfy hard real time requirements.
Summarizing, it can be said that there are two major problems with handling exceptions on core priority level 3 based on timing issues.
The first is that interrupts get delayed for a period of time that will be difficult to be estimated and is potentially infinite. This means that periodical system events occurring asynchronously will not be serviced in a timely manner and this may lead to system failure.
The second fact is that it is possible to extend the execution period of threads indefinitely via software exceptions and private exceptions handlers meaning that higher priority thread will be delayed although they should be executed.

Both cases can be described as priority inversions. The first case is a representative for priority inversion on the hardware level based on the core priority levels. The second case is a priority inversion on the software level based on thread priorities. They have an equally negative effect on the system. The second case is even more problematic is it is counterproductive to the kernel scheduler priority inversion prevention or handling mechanisms. The result for the scheduling of system threads is that the paradigm changes in these situations from preemptive scheduling to cooperative scheduling.
Another serious issue of this approach are exceptions themselves. Exceptions within exception handlers cannot be eliminated and cause unrecoverable events. The reason for this is not the code of the exception handler itself which might be exception free.
But the system state can be responsible for exceptions that are not the fault of the programmer. In order to avoid these exceptions it would be necessary to always properly align data, disable all MMU functions like caching and memory protection while executing the exception handler.

4.2.3. Alternative event flow of exception handling

Exception handlers switch to lower core priority

There is a different approach to handling exceptions than processing the exception handler on core priority level 3. It is necessary to differentiate between the two different concepts of eCos used to handle events. The first system is the virtual service routine VSR that is used for asynchronous as well as synchronous events in the system.
The second system for synchronous events is the exception handler that is provided by default by the HAL. It can be replaced by different handlers though. The HAL however will only provide one entry point for all exception VSRs that can be handled externally. This entry point is a callback to the kernel.
Based on this information there are two different points in the system where a solution could by applied.

Responsibility with exception handler

The first point is the exception handler itself. It would now be the responsibility of the programmer of exception handlers to ensure that it is not executing at the core priority level 3.
This approach has the benefit that important exceptions will be handled with the interrupt system disabled which already provides a basic scheme of synchronization.
However, this approach also has serious shortcomings. The most important is that it doesn't really solve the problem of predictability. It is still not possible for the HAL to estimate average and worst cases as some exception handlers might not leave the core priority level prior to finishing the handling.
The second problem is connected to portability. It is not possible to simply port an application to a different architecture as the exception handler itself is not architecture independent. The handler requires knowledge about how to handle and leave exception states.
This information should all be encapsulated by the HAL and not be visible to any layer above the HAL.

Due to these strong arguments the HAL will not follow this approach for external exceptions. Some exceptions handled by the HAL only without notifying upper layers, such as caching exceptions, might apply this scheme though. As it is possible for the HAL to estimate the execution time of its code.
Due to page locking the pages of the HAL will always be available to the MMU. In addition the page replacement strategies of the HAL will also ensure that other MMU related exceptions like page multiples and insufficient rights will not arise.

Responsibility with virtual service routine (VSR)

This approach allows the VSR to switch to a lower core priority level at some point during its executing. It should do so prior to calling the exception handler.
This means that the exception handler is still a normal function call only. It does not have to know the specific requirements of event handling of the processor architecture.
There are two different approaches to this problem the first is to switch to a well defined lower priority level for handling the exception as suggested by Analog Devices.
This approach cannot be applied to the BLACKfin architecture. The reason are interrupt service routines. The exception might occur within an interrupt service routine prior to masking the interrupt source. In addition it would be necessary to handle the exception prior to finishing this service routine. If the exception is handled at a well defined fixed core priority level lower than the interrupt's priority it cannot get handled and the system will stall. The only approach would be to have the exception being executed at the highest interrupt priority level below the exception state. This might solve the problem of exceptions within exceptions but not the priority inversion problem which would still persist. In addition to that it might not be possible to handle more than one exception at any one time if self-nesting of interrupts is not enabled. This solutions is not applicable.
The second solution is to leave the exception core priority level 3 and return to the priority level causing the exception. The exception handler will now execute with this priority.
This approach assures that the offending code will not continue in its execution prior to actually handling the exception. It also allows events with higher core priority than the offending event to interrupt the exception handler itself.
In addition a low priority thread causing an exception is still subject to scheduling.
The execution time required for the exception handler will be counted towards the execution time of the thread causing it.
Another benefit of this approach is that exceptions can be nested as long as the stack is large enough to hold the context of the thread for each exception.
Furthermore, exceptions only affect the stack of the thread causing the exception itself.
Once the exception handler has finished its task it will return to the virtual service routine that will then return the control to the thread causing the exception.
The demonstrated approach is portable as the exception handler is independent from the underlying architecture. In addition it makes exceptions thread local and minimizes their impact on the entire system. It also allows exceptions to be nested without causing double faults and unrecoverable events of the processor that would halt the entire system.
The exception handling described here also solves the problem of priority inversion for both cases described.

In case of hardware priorities the exception handler will execute on the core priority level of the offending code and allow all code with higher core priority to execute as if no exception had been caused.
In case of software priorities the exception handler will run in the offending threads context and not prevent the scheduler from working as the timer interrupt will still be served.
This means that a lower priority thread is still subject to scheduling and can be preempted as normal.
Another benefit is that the exception handler itself is interruptible. In case that synchronization is required all requirements imposed on ISR's do apply. This means that no scheduler interaction can be done in an exception routine unless the handler ensures that the scheduler itself is not locked. This restriction of the exception handler is not newly introduced by the BLACKfin HAL into eCos. On any system an ISR can cause an exceptions which will implicitly impose the mentioned restrictions on the exception handler.

4.2.4. Exception context

The context of a thread has been described previously. Due to the structure of eCos and the calling conventions for exception handlers the threads context as described for context switching contains the information required for exception handling.
For that matter it was necessary to include the ICPLD_FAULT and DCPLD_FAULT registers into the context as they describe exactly the location of the offending instruction or data.

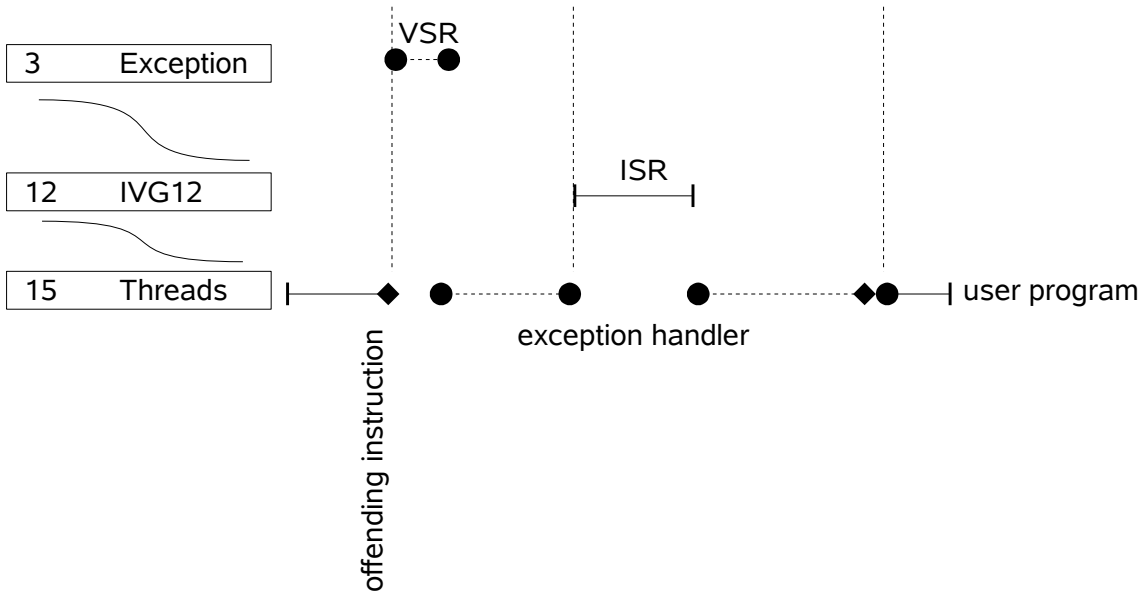


Illustration 9: concept of alternative exception handling

4.3. Interrupt handling

Interrupts are events that occur asynchronous to the instruction stream of the thread that is currently executing. They are usually used to notify the processor of system events.
The major difference between exceptions and interrupts is that interrupts are asynchronous events while exceptions are synchronous to the instruction stream.
This can be most easily seen with a small example on the BLACKfin.
While a user raised exception will be handled immediately and no instruction following the excepting will be executed a user raised interrupt will be served differently.
An interrupt might not be served immediately due to it being masked at the interrupt controller. So the thread raising the interrupt will continue to run.
This chapter will cover the interrupt system of the BLACKfin and based on the structure describe two different ways of mapping these interrupts to eCos interrupt events.
It will further describe setup and housekeeping tasks associated with interrupt handling.

4.3.1. General interrupt system

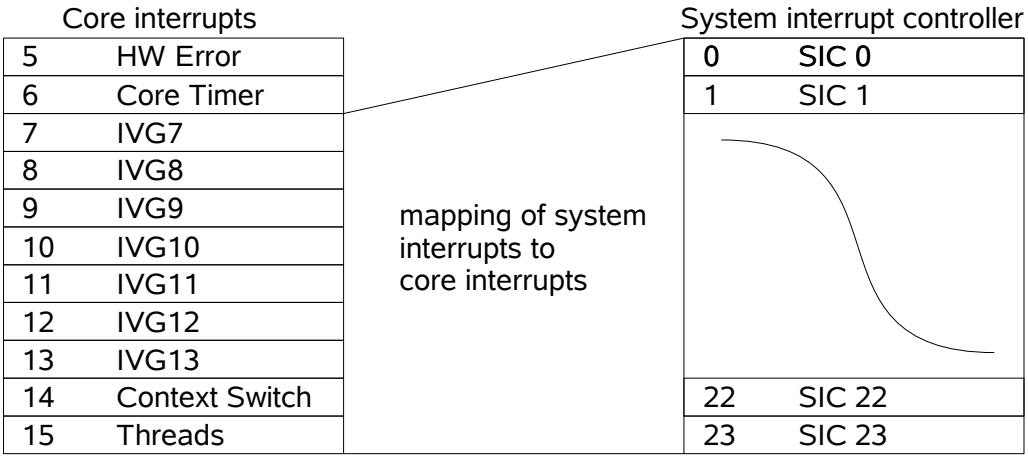


Illustration 10: layered interrupt system of the BF-533

As shown in illustration 10 the BLACKfin implements an interrupt system consisting of a core interrupt controller and a system interrupt controller. While the core interrupt controller knows the same 16 events on all BLACKfin family members the system interrupt controller may differ. The BLACKfin 533 that is used for this thesis can differentiate 24 inputs on the system interrupt controller. These 24 different inputs are now mapped to one of the available core priority levels. For all BLACKfin processors the core priorities 7 to 15 can be used to enter the core. Both controllers have their own independent mask register to unmask or mask interrupt sources. Depending on the system configuration masking or unmasking of a core interrupt might allow more than just one interrupt source from the system interrupt controller to enter the core.

The mapping between the inputs of the system interrupt controller and the core priority interrupts can be changed. This allows a flexible configuration based on the needs of the current application. eCos itself does not know a layered interrupt system and abstracts the real implementation of the system to one continuous set of interrupt sources.

4.3.2. Application cases

I will now describe two different cases in which applications need a different set of system interrupts to be serviced in order do their job. Based on the findings of the needs of the two cases I will further develop the mapping of BLACKfin interrupts to eCos interrupts.

Case 1: The simple case

In this case a normal control application will monitor a system. It will acquire its information via the UART connection as well as the programmable flag inputs.
In addition it will then distribute the results via an ethernet connection to a host computer.
In order to do the task it will also need two programmable timers and one DMA channel.
Each of the sources or sinks for information might have multiple system interrupts attached to it, as it is the case for the UART controller. The system however will handle all system interrupts belonging to one of the devices in a single service routine as it will be possible to determine the cause by interrogating the device.
Based on these findings there will be a maximum of 6 interrupts that need to be used.

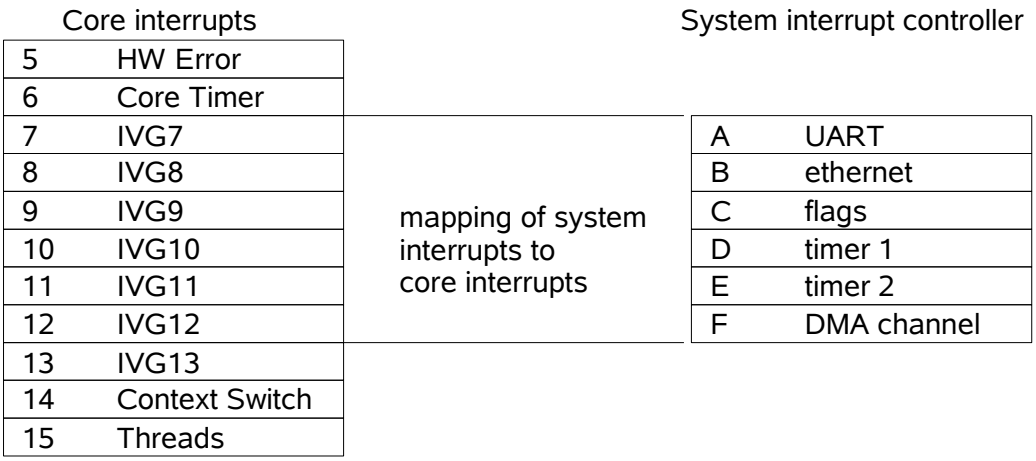


Illustration 11: case 1: simple interrupt mapping

As shown in Illustration 11 it is possible to map the interrupt sources from the system interrupt controller directly to a core interrupt.
The system configuration for this case will unmask the interrupt sources that are actually required by the application and map them to the appropriate core interrupts. All other interrupt source are masked. This means that a mapping of the core interrupt priorities is sufficient because as only one source is mapped to each core interrupt. The core interrupt taken will be identify the source. A direct access to the system interrupt controller would not yield a benefit to the current situation.

In contrast a direct access to the system interrupt controller interrupts as eCos interrupts will make the application more complicated. The UART for example does actually consist of three different interrupt inputs on the system interrupt controller. As the interrupt service routine will interrogate the appropriate registers of the UART controller it is not essential to have these interrupt sources mapped individually.
If the HAL would provide a mapping of the system interrupts to eCos interrupts the application would have to register more than 6 ISR's and possibly DSR's.
In addition providing a separate eCos interrupt for each system interrupt also means that the HAL has to determine the system interrupt causing the core interrupt. This would mean additional work by the HAL and would increase the time between the interrupt being raised and the ISR actually being serviced. So the interrupt latency would be increased.
For this application case the simple mapping of system interrupts to core interrupts is sufficient.

Case 2: The complicated case

In this complicated case the application does not just require the UART, ethernet, programmable flags, one DMA and two timers. This time the application requires all three timers, the UART, ethernet, programmable flags, PPI, SPI, two SPORTs and the real time clock.

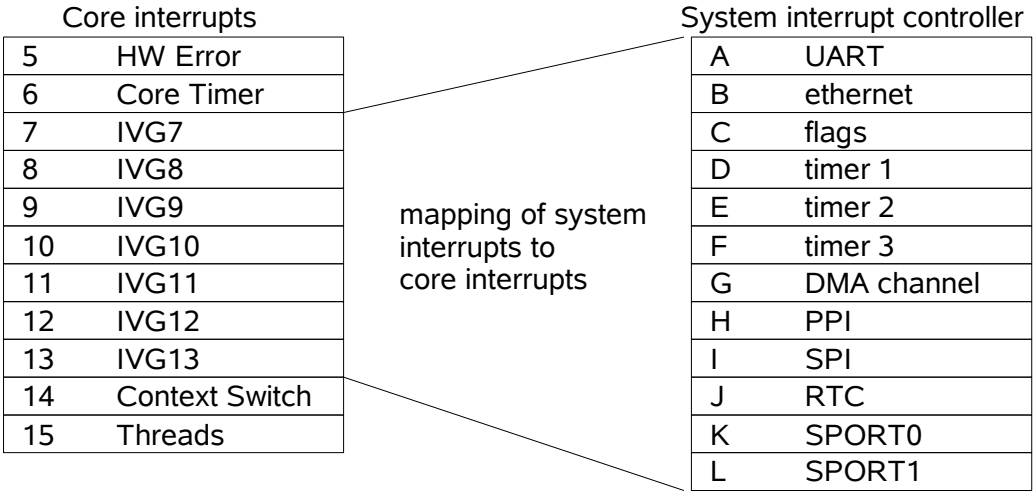


Illustration 12: case 2: the complicated case

Illustration 12 shows the problem associated with this many interrupts being used by the application. It is not possible to map just one source to one core interrupt. In case of eCos each of these sources will represent a separate device. Each of these devices will be capable of using interrupts. And eCos expects an interrupt for each of the devices shown. The problem is that the core interrupt controller does not provide enough interrupts so that the HAL can provide a separate interrupt to eCos for each of the devices.
So instead of mapping the core interrupts into the system the HAL should map the system interrupts into the system above the core timer interrupt.

In case the solution of case 1 is used the system would not work correct. As each device will register its interrupt handler to the system and many devices have to be mapped to the same core interrupt. This will mean that a device B can override the handler of a device A that had been registered earlier. The device A will no longer work and the interrupt service routine of device B would get called for event of the device A too.
A better approach would be to map the interrupts of the system interrupt controller into the set of eCos interrupts. In this situation each device would have its own eCos interrupt.

4.3.3. Interrupt mappings

Simple interrupt mapping

This interrupt mapping satisfies the requirements laid out for case 1. It will only map the interrupts of the core interrupt controller to eCos interrupts. The interrupts of the system interrupt controller will not be directly accessible. This approach allows a short interrupt VSR prior to calling the ISR for the appropriate interrupt. It still allows multiple sources of the system interrupt controller to be mapped to the same core interrupt but it will delegate the responsibility for deciding which system interrupt is the cause to the ISR.
This approach is applicable for situations described in case 1. Most applications require just a small subset of the system interrupts that is small enough so that each source can be mapped to a single core interrupt. Applying this scheme will save the overhead of determining the system input causing the interrupt as the source is already implied by the core interrupt taken.

Extended interrupt mapping

The extended interrupt mapping is applicable for all cases just as the simple mapping is. But it is most suitable for applications that require more interrupt sources from the system interrupt controller than there are core interrupts available.
In this case no core interrupt above 6 will get mapped to eCos. The inputs of the system interrupt controller will be mapped beginning from interrupt 7 instead.
The reason why the core interrupts from 7 to 15 are no longer mapped can be explained by the way these interrupts get handled by the HAL.
The extended interrupt mapping requires the HAL to discriminate between the different inputs that are mapped to the current core priority level and select one to be served.
As the core interrupts are now only used to determine the priority of the system interrupts in relation to each other it is not necessary to map them to eCos interrupts.
In addition mapping these interrupts might cause problems.
An application programmer could accidentally register an ISR for one of these interrupts. This would mean that the HAL's handling service would get replaced. If that happens the system interrupts assigned to this core priority will not be served via there corresponding eCos interrupt anymore.

Comparison of the two mappings

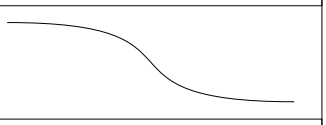
simple mapping			extended mapping	
5	HW Error	fixed as both events directly enter the core	5	HW Error
6	Core Timer		6	Core Timer
7	IVG7		7	SIC 0
8	IVG8		8	SIC 1
9	IVG9			
10	IVG10			
11	IVG11			
12	IVG12		29	SIC 22
13	IVG13		30	SIC 23

Illustration 13: BLACKfin to eCos interrupt mappings for BF-533

The illustration 13 shows a direct comparison between the two interrupt mapping schemes that will be provided by this HAL.
As it can be seen the interrupts 5 and 6 are equal in both mappings. This behavior is due to the fact that both events enter the core directly at the core interrupt controller while all other interrupts shown enter the core via the system interrupt controller.

4.3.4. Services required for the interrupt system

The BLACKfin HAL has to provide additional services than just mapping the interrupts.
The HAL will also provide a default interrupt VSR. This default VSR will be attached to the VSR number corresponding to the interrupt number.
A VSR as described in the section about exceptions is a virtual service routine and is the entry point for any event in eCos.
Furthermore, the HAL must provide a default interrupt service routine. The task of this routine is only to return and declare the interrupt to be served.
In addition to services related to processing interrupts there are also services required for configuring the interrupt system.
The HAL has to provide functionality to mask and unmask interrupt sources.
These services are provided as macros. The implementation of the macros might differ based on the interrupt mapping that has been chosen.
In addition the HAL will provide services to enable, disable and restore interrupts.
It will also provide macros that allow the assignment of system interrupts to core priority levels at runtime.

4.4. MMU and caching

The BLACKfin processor has a built-in memory management unit that supports page based protection and caching options. In order to use either feature the MMU must be enabled.

4.4.1. Caching

This thesis will provide a basic support for the memory management unit on the BLACKfin processor to be able to support caching for this architecture.
It is necessary to determine which parts of the caching system belong to the architecture and which should be placed into the variant HAL.
Different members of the same processor family typically differ in their cache sizes as well as functions the cache supports. In case of the BLACKfin processor family the functionality differs only in case of instruction caching. While some members have only one bank for instruction cache others support two. This will result in a different structure of the register controlling which banks of the cache are actively used as cache or SRAM.
Based on these information most parts of the BLACKfin cache support can be placed in the architecture HAL. While some macros regarding cache enable and disable will need to be included on a conditional basis only. This means that according to the eCos strategy it will be possible for a future variant HAL supporting a different member to override the default macros. In addition the cache sizes will be placed in the variant HAL as they will most likely be different. As mentioned earlier it is necessary to enable the memory management unit of the processor in order to use caching.
The processor supports two different caching strategies known as write-back and write-through. eCos does also support the selection of either strategy. The HAL will use the selection of the common HAL parts of the configuration. In addition to these basic options the BLACKfin does also support four different page sizes. The user will be able to choose between the supported size of 1 kB, 4 kB, 1 MB and 4 MB for instruction and data cache separately. The BLACKfin does also provide optional support for cache line allocation on write when operating in write-through mode. The HAL will also support the selection of this option.

4.4.2. Pages and page replacement

In order to use caching as well as protection mechanisms it will be necessary to provide a paging strategy. This is necessary as the BLACKfin processor does not support traversing a page table in hardware. It only features one table with 16 entries each for instruction and data accesses. Given the maximum size of a page of 4 MB it will be impossible to cover the entire memory range with one set of pages only. The maximum amount of memory that can be covered will be 64 MB.
The option chosen for this thesis is not to use a separate page table in memory.
All functions that are influenced by the strategy will be separated from the general handling code for page misses.
The code will use a function to determine the table entry to replace. This function can apply its own victimization strategy for table entries. In addition another function will be used to determine the properties of the page for the specific address to be entered into the table.

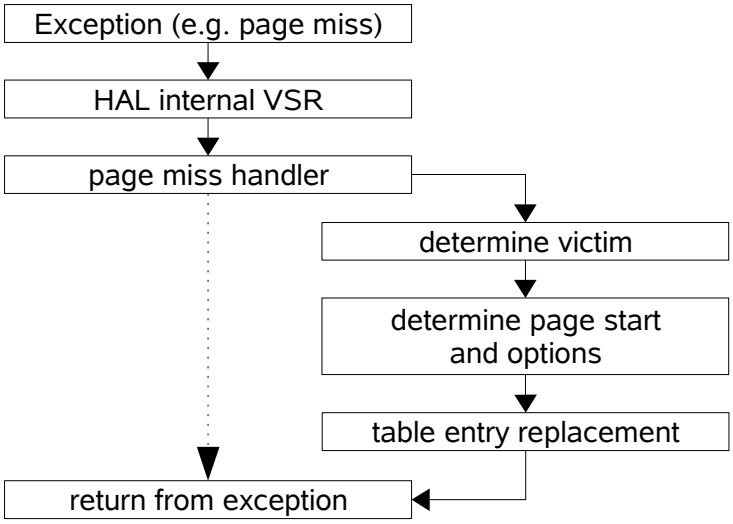


Illustration 14: page miss handler

All functions should be declared inline to reduce unnecessary function calls to these code fragments. The event flow of a page miss exception is shown in illustration 14.
The illustration shows the logical flow if an page miss exception. The exception will occur and be serviced via the VSR and the page miss handler before returning to the thread. The actual flow within the page miss handler though it to determine a victim, followed by finding the new page's properties before replacing the current entry in the MMU table.
This scheme is extensible as different strategies can be applied to victimization as well as paging. It would be possible to apply a sophisticated algorithm to determine a victim to be replaced in the table. In addition a page table based approach could be used to determine the page properties. This is especially useful if a fine granular configuration is desired. The approach with a page table and potentially different page sizes also has a downside.
In case of mixing different page sizes the case of accidentally adding two different pages partly covering the same part of memory could arise. In this case the handler for a page multiple exception would have to be used too.
This eviction could be based on different properties, a ranking within the page table or the page size or even the page properties.

One example would be the most-restrictive strategy. In this case the page with the most restrictive rights would survive and the remaining page would have to be invalidated.
This might be particularly useful if eCos would support a more detailed view on stacks. In this case it would be possible to place small protective pages with no write access around the stacks boundaries to prevent stack overflows for example. A page could be placed at the lower end of the stack - as stacks are growing downwards. In case an application would attempt to write into this area a page multiple exception could be the result.
The handler could now determine the kind of access and allow or disallow the action.
This approach would allow stack boundary checks for stacks that are not multiples of the page size.

Paging strategy

The page generation and victimization strategy chosen for this implementation is due to time constraints of the thesis itself more simplistic.
The victimization will follow a simple round robin strategy. Always choosing the next page in the round only requires a counter that will be incremented or decremented modulo the table size.
The eviction strategy will not select table entries that are marked as locked.
The page generation strategy chosen will also not involve a page table to be held in memory. The pages will be classified based on their location in memory and will than yield the caching and protection properties based on the location.
This means that the strategy is limited in its flexibility compared to the page table approach but it does not require further memory for the HAL and will not require the HAL to ensure proper access to the page table location itself.

Page properties

The HAL will implement a basic scheme of determining the caching options of a page.
A page will be cached in case it does reside in SDRAM. All pages outside the SDRAM will not be subject to caching. The reason for this criteria is that the remaining part of memory is predominantly reserved for input/output devices and system and core memory mapped registered. As these registers and memory locations might change due to core and device as well as DMA activity caching is not desired as it might lead to consistency problems between the cache and the memory.
In addition eCos can further specify the access rights to the pages. As eCos operates in supervisor mode only no read or write access is necessary for user mode.

5. Implementation

5.1. Exception handling

5.1.1. Exception trampoline

The exception trampoline has the purpose of determining the exception cause and selecting the appropriate handler that has been registered in the VSR table.
In order to fulfill this task the trampoline code has to query the SEQSTAT register. This register is a system register holding the exception cause in the least significant 6 bits. This means that the BLACKfin processor caters for a maximum of 2^6 or 64 different exception causes. 16 of these causes are software exceptions and the remaining 48 causes are raised in hardware as result of certain events.
Even though not all possible exception numbers are currently used it is necessary to map every cause possible in the SEQSTAT register to a VSR table entry. This mapping will result in some places in the VSR table to be essentially unused but it also drastically simplifies the mapping between SEQSTAT cause and VSR location.
The simplification will yield speed improvements in the code that determines the VSR handler.

```
      hal_bfin_exception:
1:    SSYNC
2:    USP    = P0             // save P0 in USP, as no stack is available
3:    P0.L   = __hal_exception_temporary_frame
4:    P0.H   = __hal_exception_temporary_frame
5:    [P0 + 0x8 ] = R0        // save R0, R1 and P1 because they are used
6:    R0          = ASTAT     // ASTAT cannot be stored directly
7:    [P0 + 0x4 ] = R0        // save the flags on the stack
8:    [P0 + 0xC ] = R1
9:    [P0 + 0x10] = P1

      // trampoline code to determine exception cause and entry point into
      // handler code to go here, the code does only use R0, R1 and P1
      // and it will place the entry point at P0+0x0

10:   P1     = [P0 + 0x10]    // restore all registers P1, R1, R0, ASTAT
11:   R1     = [P0 + 0xC ]
12:   R0     = [P0 + 0x4 ]
13:   ASTAT  = R0
14:   R0     = [P0 + 0x8 ]
15:   P0     = [P0]           // restore
16:   JUMP    (P0)            // jump to the exception handling routine
```

Code 1: exception trampoline fragment

The exception trampoline does require working registers because there are no memory only operations supported by the BLACKfin. This means that source and destination of an operation will have to be stored in the register file. In order to to use certain registers the current value has to be saved to be restored later. The reason for this is that an exception is not a normal function call but another type of synchronous event that can occur without an explicit interaction by the programmer. This means that all registers of the program must be preserved.

As shown in code segment 1 the trampoline is saving registers that are used to a location called __hal_exception_temporary_frame. The reason for saving the data at this location and not using the threads stack is because the stack might not be available.
Due to the structure of eCos the HAL is not able to determine the location of the stack and with this incapable of locking the appropriate pages if either protection or caching is used. The threads code might behave in a way that will lead to a victimization of the page containing the threads stack immediately prior to raising the exception.
In case the exception trampoline would now try to place its data on the stack it would raise a second exception. As described in the argumentation about the exception handling this is an undesired event and has to be avoided. The special location used is placed in the HALs context and with this in a locked location. This means that the page descriptor for the memory location will always be available ensuring that no further page miss exceptions are caused.
The code is also showing a convention by the HAL that the USP register must not be used at any time by any piece of code except for the exception trampoline and VSR while in an exception state of the processor. The latter condition of the exception state is necessary for synchronization. Due to the fact that the USP register is not saved in the trampoline and the locations to store the working registers are fixed this piece of code is not reentrant. This condition however is not a limitation by the HAL as an exception is not reentrant by convention of the BLACKfin architecture. In case that another exception would occur the HAL would not only override the saved working registers and USP but the architecture would also override the RETX, ICPLD_FAULT and DCPLD_FAULT registers in addition to the SEQSTAT register. So this code is not required to be reentrant.
The trampoline code has no precondition except for only being called via the event vector table of the processor if in an exception state.
The trampoline has the post conditions that it will jump to the entry point of the VSR for the current exception and leaving this entry point in P0 and the original P0 in USP.
It is the responsibility of the VSR to restore the original value of P0.
In order for the trampoline and the entire system to work properly a third post condition is imposed by the trampoline that will not be checked but must always be true.
The page containing the entry point of the VSR as well as all further code of the VSR that is executed in the exception state must be accessible with the current set of ICPLD descriptors in case the memory management unit is in use. This condition is always met by the default virtual service routine provided by the HAL

5.1.2. Exception VSR

The exception VSR plays an important role in the implementation of the system as it is responsible for switching back to the core priority causing the exception while saving enough information to facilitate the exception handler to properly handle the event.
As described in the arguments about where exceptions should be handled all normal exceptions that do not need to modify the page descriptor tables will be handled on the core priority level that has caused the exception. Exceptions that need to change the memory management unit's internal tables however will be processed on the exception level as this includes an implicit synchronization. The exception handler has as preconditions the postconditions of the exception trampoline. This means that the VSR will start processing in the exception state of the processor.

```
        __default_exception_vsr:
1:      P0  = USP   // restore P0
2:      USP = SP    // save stack pointer
3:      SP.H = __hal_exception_temporary_frame_stack
4:      SP.L = __hal_exception_temporary_frame_stack
5:      [--SP] = (R3:0,P2:0) // save working registers
6:      [--SP] = RETS
7:      R0 = USP

8:      CALL _cyg_hal_assure_sp_access

9:      RETS   = [SP++]
10:     (R3:0,P2:0) = [SP++]
11:     SP = USP
12:     savecpustate
```

Code 2: simplified default VSR entry

The first task of the VSR should be to restore the threads context which only means that P0 needs to be restored. Based on this the exception VSR will have to save parts of the threads context and additional information about the exception cause prior to handing over the control to the exception handler. Whether the VSR will continue processing on the current exception priority level or not is up to the VSR.
As shown code segment 2 for the default VSR entry the VSR has to perform another important task prior to saving the threads context which is the last instruction called *savecpustate*. The VSR has to assure that the stack of the thread is accessible.
It will do so by calling the _cyg_hal_assure_sp_access function. This function will put the appropriate page descriptors in place for access to the stack. Only the page descriptors required to store the threads context will be guaranteed after this call returns.
The default VSR supplied by the BLACKfin HAL will switch to the core priority causing the exception. As VSR's can be replaced by user programs in order to directly interact with the exception it is possible to apply a different strategy by user handlers.
All VSRs however have in common that the postconditions after finishing must be a restored threads context and the switch back to the priority level of the thread that originally caused the exception.

```
1:    P0.L = __default_exception_vsr_continue
2:    P0.H = __default_exception_vsr_continue
3:    RETX = P0
4:    RTX

      __default_exception_vsr_continue:
5:    R0 = FP     // first argument is pointer to context
6:    CALL  _cyg_hal_exception_handler
7:    EXCPT 0

      __default_exception_vsr_return:
8:    P0 = USP    // P0 saved by exception trampoline
9:    restorecpustate
10:   RTX         // return to the interrupted task
```

Code 3: exception VSR continue

Code segment 3 is now illustrating the further processing of the exception.
The VSR will return from the exception state within its own code and continue processing at _default_exception_vsr_continue. As outlined in the concept the exception handler will be invoked. But after the exception handler returns the implementation differs from the concept. The reason for this is the way the BLACKfin is handling exceptions and the storage of return addresses. Due to the special register for the return address it is not possible to use a normal return from subroutine RTS instruction.

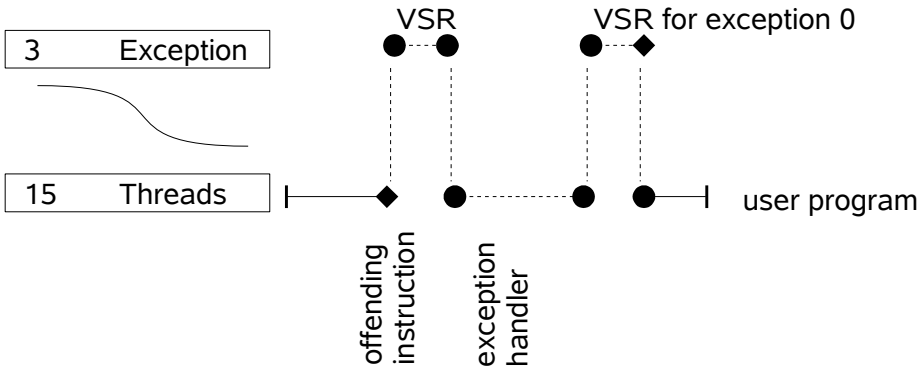


Illustration 15: implementation of alternative exception handling

This instruction assumes the return address in register RETS which cannot be overridden as the current value must be preserved and the default VSR would violate the postconditions. It is further not possible to do a JUMP as all registers that are available to hold the sources cannot be changed. The only possible way to return to the thread is by raising another software exception. This exception 0 is reserved for this purpose only and cannot serve any other purpose. The trampoline will jump to the entry point for exception 0 which is located at the label __default_exception_vsr_return. The new graphical representation of the exception handling can be seen in illustration 15.

5.2. Interrupt handling

5.2.1. Interrupt trampoline

Based on the configuration chosen there will be two different trampolines. The first trampoline code will always be used for the hardware error and core timer interrupt.
It will also be used for interrupts IVG7 to IVG13 in case the simple interrupt system is used. In this case the core interrupt number taken is equal to the index into the VSR table.
The handling service starts off with a common part of putting the current interrupt number directly into the register P0. This is done via a macro that is instantiated once per core interrupt. This approach has been chosen to reduce time required to determine the highest core priority.
hal_bfin_interrupt_common code will use the interrupt number in R0 to load the entry point of the VSR table.
In case of the extended interrupt system the code will call into *hal_bfin_sic_interrupt_common* to determine the interrupt number on the system interrupt controller.

```
1:    R1     = R1 ^ R1
2:    R1     = 0x7
3:    R1     = R0 - R1  // we now have the interrupt lane
4:    R2.H   = hal_interrupt_sic_assign_mask
5:    R2.L   = hal_interrupt_sic_assign_mask
6:    R1     <<= 2
7:    R2     = R2+R1
8:    P1     = R2
9:    R0     = [P1]     // R0 now contains the sic assign mask
10:   P1.H   = 0xFFC0   // load the current SIC ISR register
11:   P1.L   = 0x0120
12:   R1     = [P1]
13:   R0     = R1 & R0  // eliminate all SICs not assigned to core priority
14:   P1.H   = 0xFFC0
15:   P1.L   = 0x010C
16:   R1     = [P1]     // get the current SIC MASK
17:   R0     = R1 & R0  // only unmasked SICs for this core priority left
18:   CC = R0           // should never be 0
19:   IF !CC JUMP hal_bfin_sic_interrupt_common_abort
20:   CALL _hal_lsbit_index
21:   R1.H = 0x0  // R0 now contains the interrupt number at the SIC
22:   R1.L = 0x7  // R0 should never be -1
23:   R0   = R1 + R0
```

Code 4: extended interrupt system trampoline fragment

The code fragment 4 shows the part of the extended interrupt trampoline that has to be inserted to determine the VSR index. It assumes that the core interrupt priority is stored in R0. Based on that it will determine the assignment number by subtracting 7.
This number is necessary to determine the mask of interrupts on the system interrupt controller SIC that are assigned to this core interrupt. In order to allow the HAL to do this task efficiently the mask will be kept and updated by the HAL whenever the assignment of an SIC to core priority level is changed. This allows the mask to be determined with only a few instructions.

In case the HAL would not keep the mask for each core interrupt it would be necessary to query all assignment registers and build this mask as needed. The main goal of the trampoline however is to be fast to reduce the time the interrupt system is temporarily disabled. In addition only software can change the assignment and this means that only at the time a change is done the masks have to be updated as they are invariant for the remaining time.
In addition to masking off all interrupts that are not assigned to this core priority level it is also necessary to mask off all interrupts that are actually masked.
The system interrupt controller does not offer a pending register. The register of the SIC will have a bit set for every source that is asserting an interrupt even if the source is masked. Masked sources don't lead to a notification of the core interrupt controller though. However, it is necessary to get the set of interrupts that are assigned to this core interrupt and unmasked.
After this has been done the trampoline will use a HAL internal function hal_lsbit_index. This function takes a mask and determines the bit index of the least significant bit set in the mask. This means that the HAL is prioritizing sources with lower input numbers on the system interrupt controller. It would also be possible to use the highest priority bit or to use a more sophisticated algorithm. The main goal however is still to be fast. The more sophisticated the algorithm for selecting the interrupt will be the more processing time it will consume. The hal_lsbit_index function takes about 29 core cycles to complete disregarding which bit positions are set.
The reason for choosing the least significant bit over the most significant was to follow the same approach applied by the core interrupt controller. As mentioned earlier the core interrupt controller prioritizes inputs with a lower index.
Eventually the index determined by the trampoline has to be shifted to start indexing from location 7 in the VSR table as all other entry points of the core priority controller will directly index into the table.
After that both interrupt trampolines will simply determine the entry point of the VSR from the VSR table and jump to the VSR. While doing so they will leave ASTAT, R0 and P0 on the stack. R0 will contain the eCos interrupt number which is equal to the VSR table index and P0 will contain the entry point of the VSR.

5.2.2. Enabling and disabling the interrupt system

eCos provides means for enabling and disabling the interrupt system. This is done for instance by some critical parts in the kernel but it can also by user programs. The macros HAL_DISABLE_INTERRUPTS and HAL_ENABLE_INTERRUPTS as well as HAL_RESTORE_INTERRUPTS are responsible for this task.
This approach itself is not problematic but the implementation chosen by the kernel poses a problem for the HAL. The kernel does always assume that masking and unmasking interrupts is a non interrupt safe action. So in order to be interrupt safe the kernel will first disable interrupts, then mask or unmask the source before restoring interrupts.
The problem for the BLACKfin HAL is that there is no global interrupt enable/disable flag. The processor provides instructions for enabling and disabling interrupts but they do not work exactly as desired by the macros. The only task these macros should do is globally enable or disable the interrupt system while keep the interrupt mask untouched. Unfortunately this approach is not possible. Whenever the instructions STI for set interrupts or CLI for clear interrupts are used the core interrupt controller mask is altered. When the CLI instruction is used the mask is cleared. The STI instruction will set the mask with the given mask.

This leads to a lost update problem if the kernel wants to mask or unmask an interrupt. It will first clear interrupts which will yield the old state. It will then mask or unmask the interrupt before restoring the old state. As the state would include the old interrupt mask the newly updated mask would be lost. An interrupt could never be unmasked or masked by the kernel.
In order to avoid this the HAL has implemented a work around for this problem that is applicable for all situations. In addition the masking and unmasking operations themselves are interrupt safe.
The code in fragment 5 shows the disable macro for interrupts. So far the macro does not really look different from the normal approach of using CLI. Except that the code following the CLI instruction will mask off specific bits. The only bit remaining in the mask is bit number 15. The reason for leaving bit 15 in the mask is due to its special purpose function. Core interrupt number 15 is reserved for threads and raised once only during startup prior to configuring interrupts.

```
      asm volatile (
1:    "CLI R0;"
2:    "R1.H = 0x0;"
3:    "R1.L = 0x8000;"
4:    "R1   = R1 & R0;"
5:    "%0   = R1;"
      : "=r"(_old_) : : "R0", "R1"
      );
```

Code 5: disable interrupts

After that no further interrupts of level 15 must be raised. The bit for this interrupt in the mask will be used to determine whether interrupts were actually enabled or not.
This is necessary as disabling all interrupts should only be reversed by enabling or restoring the enabled mask. But the masking and unmasking operations of specific interrupts should not actually allow them if interrupts are disabled.
As disabling the interrupt means that the interrupt mask is cleared unmasking of interrupts would damage the cleared mask by setting specific bits. This would allow these interrupts even though interrupts should be globally disabled. For that purpose bit number 15 will serve as an indicator whether interrupts are globally disabled or not.
In case bit 15 is zero interrupts are disabled and no masking and unmasking is allowed to change the hardware register containing the mask. The operations will be done in the software shadow register *_hal_intr_mask_applied* only.
All masking and unmasking operations that affect the core interrupt controller mask will be done using the mask stored at this location. Depending on the system state the mask will than be used to update the hardware based mask.
Code fragment 6 is showing the restore macro for interrupts. It expects the old state in _old_ and will determine based on that value if the interrupt masked should be cleared or restored.
The macro is very simple and uses the conditional move operation (line 10) to avoid jumps.

```
      asm volatile (
1:    "CLI R1;"
2:    "R0 = %0;"
3:    "P0.H = _hal_intr_mask_applied;"
4:    "P0.L = _hal_intr_mask_applied;"
5:    "SSYNC;"
6:    "R2   = [P0];"
7:    "SSYNC;"
8:    "R1 = R1 ^ R1;"
9:    "CC = R0;"
10:   "IF CC R1 = R2;"
11:   "STI    R1;"
12:   "SSYNC;"
      : : "r"(_old_)
      : "R0", "R1", "R2", "P0" );
```

Code 6: restore interrupts

The mask to be loaded is zeroed by default (line 8). Based on the value old this mask will be replaced with the software shadow mask (lines 9, 10) prior to updating the core interrupt mask register (line 11).
The conventions shown here do influence the mask and unmask macro for specific interrupts too.
The following section will have a closer look on how these macros operate in order to provide the behavior required by eCos.

5.2.3. Masking and Unmasking of interrupts

The masking and unmasking macros are important to the system as they do allow a specific interrupt to reach the core and the corresponding ISR to be called.
The macros will also have to follow the conventions shown in the previous section.
This does specifically mean that they must not change the hardware mask unless interrupts are globally enabled. In addition, they either have to ensure or assume that bit 15 is always set in the software shadow mask.

Simple interrupt system

Code fragment 7 shows the unmasking macro. The masking macro works analogous and is omitted.
All masking and unmasking of interrupts is interrupt safe. This means that the masking and unmasking operations are critical sections.
The synchronization mechanism used is to temporarily disable interrupts (lines 1, 15).
Further the macro will load the current mask from the shadow register and set the bit representing the interrupt number in this mask. (lines 2 to 8)
Following this the macro tests whether bit 15 in the current hardware mask is set.
If so than the hardware mask will be replaced with the software shadow mask. (lines 9 to 13)
As masking operations are always done on the shadow register first no updates can be lost due to overriding the hardware mask.
It is strictly prohibited for any program to bypass the HAL by setting registers within the HAL's responsibility without using the appropriate HAL macros. Prior to activating the interrupt system the new shadow mask value will be updated in memory.
The masking macro will work analogous to the one shown with a different sequence of instructions replacing line 8.

```
      asm volatile (
1:    "CLI R1;"
2:    "R0   = R0 ^ R0;"
3:    "P0.H = _hal_intr_mask_applied;"
4:    "P0.L = _hal_intr_mask_applied;"
5:    "R2   = [P0];"
6:    "R0   = 0x1;"
7:    "R0   <<= %0;"
8:    "R2   = R2 | R0;"
9:    "R0.H = 0x0;"
10:   "R0.L = 0x8000;"
11:   "R1   = R1 & R0;"
12:   "CC   = R1;"
13:   "IF CC R1 = R2;"
14:   "[P0]   = R2;"
15:   "STI R1;"
16:   "SSYNC;"
      : : "r"(_vector_)
      : "R0", "R1", "R2", "P0" )
```

Code 7: unmasking interrupt (simple)

Extended interrupt system

The macros for the extended interrupt system are a bit more complex but follow the same basic principal. The macro shown in fragment 7 can be found as part of the extended version and is used whenever the eCos interrupt is smaller than 7.
For all eCos interrupts with numbers greater than 6 a different approach has to be used.
The macro does not require a shadow mask register for interrupts that operated via the system interrupt controller in the extended interrupt system. The reason for this is that the system interrupt controller mask is not affected by the CLI and STI instructions.
This means that the macro can directly change the system interrupt controller mask by masking or unmasking the corresponding bit positions.
In addition any masking or unmasking operations that is performed on the system interrupt controller will be accompanied by a possible change to the core controller mask.
The same conditions as shown in code fragment 7 apply for this update to the core controller mask.
The purpose of the update is to ensure that the core priorities serving as inputs for the system interrupts are always unmasked whenever interrupts are enabled.

Code fragment 8 shows parts of the macros for unmasking interrupts in the extended interrupt system. The part shown only covers the unmasking of system interrupts which are represented by eCos interrupts with number greater than 6.
In order to determine the interrupt number at the system interrupt controller 7 has to be subtracted from the eCos interrupt number provided to this macro. (line 1)
As shown in the previous examples the masking and unmasking operations are not interruptible. This is ensured again by temporarily disabling the interrupt system.
Following this the macro will load the mask of the SIC into a working register.
(lines 3 to 6)
Lines 8 to 10 are similar to the simple unmasking just that no shadow register is required and the mask of the system interrupt controller can be updated directly.
Note that this task must be done with interrupts disabled as the interrupt source could interrupt this code after the mask has been updated at the controller.
Lines 12 to 17 show the unmasking of core priorities IVG7 to IVG13 that could be the entries of system interrupts to the core.

```
1:    internal_vector=_vector_-7;
      asm volatile (
2:    "CLI R1;"
3:    "R0   = R0 ^ R0;"
4:    "P0.H = 0xFFC0;"
5:    "P0.L = 0x010C;"
6:    "R2   = [P0];"
7:    "R0   = 0x1;"
8:    "R0   <<= %0;"
9:    "R2   = R2 | R0;"
10:   "[P0]   = R2;"
11:   "SSYNC;"
12:   "R2.H = 0x0;"
13:   "R2.L = 0x3F80;"
14:   "P0.H = _hal_intr_mask_applied;"
15:   "P0.L = _hal_intr_mask_applied;"
16:   "R0   = [P0];"
17:   "R0   = R0 | R2;"
18:   "R2.H = 0x0;"
19:   "R2.L = 0x8000;"
20:   "R1   = R1 & R2;"
21:   "CC   = R1;"
22:   "IF CC R1 = R0;"
23:   "[P0]   = R0;"
24:   "STI R1;"
25:   "SSYNC;"
      : : "r"(internal_vector)
      : "R0", "R1", "R2", "P0" );
```

Code 8: unmasking system interrupt (extended)

The lines 13 to 23 work analogous to the unmasking macro for the simple interrupt system shown in fragment 7.
The updates to the system interrupt controller will be done similar for the case of masking system interrupts.

5.3. Further issues during implementation

5.3.1. Caching with write-back mode

The BLACKfin architecture does not provide facilities to flush the entire cache.
This means that it is not possible to write all changes made in the cache out to memory with a single instruction. The only way for solving this problem would be to either flush all locations that might be in the cache or to actively search the cache for dirty cache lines. The first option is applicable only for small page sizes up to 4 kB. Above this size limit the amount of possible addresses within the page is too large. As the cache line size is 32 bytes it would be necessary to iterate over 32,768 addresses in case of a 1 MB page. This is impractical knowing that the the BLACKfin BF533 has a maximum of 32 kB of data cache. Taking the cache line size into account this would yield a maximum of 1024 different cache lines to be flushed. Iterating over each cache line would be faster if less than 32 instructions are required per iteration.
For write-through mode caching this problem does not exist as all changes done in the cache are automatically forwarded to the memory.
For that matter write-through caching will be implemented first.

5.3.2. Adapting drivers to interrupt system

Another problem that had to be tackled during the implementation was that drivers had to be adapted for the extended interrupt system. All of the devices used for this port of eCos on the STAMP BF-533 board are already supported by eCos.
The UART controller on the BLACKfin chip is PC compatible this means that the standard UART driver of eCos could be used. The driver was adapted and is no longer part of the standard UART code base. The reason for this adaption is the extended interrupt system of the BLACKfin HAL and the special considerations that had to be given to the implementation of the UART on the BLACKfin. The controller is only PC style compatible if the simple interrupt system is used and all input sources of the UART are unmasked and mapped to the same core priority. This would already require additional code for masking and unmasking of multiple system interrupts. The general driver would no longer be applicable for the extended interrupt system though. Based on these findings the a copy of the driver code has been moved to the BLACKfin code base and adapted for the situation of the UART providing two dedicated receive and transmit interrupts at the system interrupt controller.
The driver now supports both operating modes. In addition only one configuration option has to be provided for both interrupt systems. The user will only select the core priority of the UART controller rather than the specific interrupt. This approach is applicable because the core priority will be used for the trampoline code and in the simple interrupt configuration. The driver does only support both interrupt inputs of the UART to be mapped to the same core priority.

5.3.3. Memory sharing

The design of the STAMP board requires that the asynchronous memory and the ethernet chip share the same memory location from 0x20000000 to 0x203FFFFF.
While the FLASH memory occupies the entire memory range the ethernet chip only occupies a few bytes in the range of 0x20300300. The board designers have chosen to use the programmable flags to switch between the two devices. The original driver did not support this switching as it assumes to have its own unshared memory region.
In order to facilitate the driver to use the ethernet chip some wrapper functions had to be written for the original driver functionality.
These wrapper functions serve the purpose of switching to the ethernet chip prior to invoking the original driver and switching back after the driver function returns.
The code fragment 9 shows the switching in lines 1 and 3 while the original driver function is called in line 2.

```
      static void stamp_lan91cxx_poll(sc) {
1:    SWITCH_TO_ETH();
2:    lan91cxx_poll(sc);
3:    SWITCH_RESTORE();
      }
```

Code 9: sample ethernet wrapper function

In addition the device initialization function required some adjustments for both the extended interrupt system as well as setting up the programmable flags for later memory region switching.

6. Results and findings

6.1. Current state of the porting process

The new hardware abstraction layer for eCos called bfin follows the approach of splitting the functionality over three separate parts. In addition to the HAL the support for eCos is sufficient to provide configuration up to the network stack which is the highest standard configuration option. In addition RedBoot is fully functional as boot loader on the STAMP BF-533 board.

6.1.1. Architecture HAL of the BLACKfin

The architecture HAL for the BLACKfin provides support for the exception and interrupt handling. The interrupt handling is fully implemented according to the principles described in the design section of this thesis.
It further incorporates the special findings for the interrupt system satisfying the particular architecture specific handling of the global activation respectively deactivation of interrupts. Interrupts are mapped in the two described ways to eCos interrupts and properly mapped via the VSR table. The HAL does also provide the architecture specific default interrupt VSR which will handle all hardware related actions necessary prior to invoking the interrupt service routine from the ISR table. This default VSR will also interact with the scheduler as required by eCos in case the eCos kernel is used.
In addition to the parts implemented in the architecture HAL some parts of the interrupt system have been implemented by the variant HAL. In case the simple interrupt system is used most parts are implemented by the architecture HAL directly as simple system only depends on the architecture but not on the specific family member.
In the extended interrupt system the number and assignments of interrupts are implemented by the variant HAL as they depend on the specific variant in number and assignment to core devices.
The architecture HAL does further provide the full support for exception handling as described in the design section. It does implement the mapping of exceptions to eCos VSR entries and the default exception VSR. In addition the default exception VSR does provide an implementation of the alternative exception handling which defers the handling to the core priority level causing the exception rather than direct processing on the exception core priority. It considers the special requirements described in the implementation section regarding the placement and use of return address registers on the BLACKfin rather than return addresses placed on the stack. This means that the special implementation with the return from exception special software exception follows the designed event flow as closely as possible and only differs in the return from exception part. Another important part implemented in the architecture HAL is the context switching facility necessary for using the eCos kernel. In addition the context switching is provided for any program not using the eCos kernel. As the register set and the general requirements to context switching are invariant the entire context switching code is placed in the architecture HAL.
The architecture HAL does further provide general initialization macros necessary for setting up the board during startup as well as the input output device macros provided by eCos for a universal access to IO devices on different architectures using either memory mapped access like the BLACKfin or an IO mapped access.

6.1.2. Variant HAL of the BF-533

The variant HAL of the BF-533 member of the BLACKfin provides support macros for accessing the system interrupt controller required for the simple interrupt system as well as the masking and unmasking macros for the extended interrupt system.
Additionally the cache sizes are defined in the variant specifically for the BF-533 processor.
In addition to the BF-533 the variant HAL also widely supports the BF-537 as it had been used for portability testing. The testing had been done with the STAMP board containing a BF-537. For that purpose only the cache sizes and the amount and mapping of the system interrupts to devices had to be changed.
In addition to the general interrupt handling specific to these two representatives of the BLACKfin family the variant HAL also implements code required for setting up the mapping of system interrupts to core priority levels.
Furthermore, tables like the VSR table, the ISR tables for entry points, ISR data pointers and ISR object pointers required by eCos are defined in the variant HAL as they heavily depend on the specific variant.

	architecture	***variant***	***platform***
context switching	fully supported respecting architecture specific requirements for interrupt returns	not required	not applicable
exception handling	exception vectors, default exception VSR, default exception handler	exception mapping into VSR table	not applicable
interrupt handling	default interrupt VSR, default interrupt ISR, global enable/disable, masking and unmasking in simple interrupt system,	interrupt number and mapping, SIC core priority assignments, masking and unmasking in extended interrupt system	not required
diagnostic output	general setup	not required	output drivers implemented
caching	general replaceable caching macros, caching related exception handlers for miss and multiple exceptions	caching sizes	not applicable
hardware setup	architecture specific setup	variant specific setup	setup configuration for asynchronous and synchronous memory and devices required by the HAL

Table 2: implementation of functions in different HAL parts

6.1.3. Platform HAL of the STAMP board

The STAMP board being the platform of the porting efforts of this thesis is represented by its own platform HAL in eCos. The implementation provides a simple non-interrupt based driver for the serial connection used for diagnostic output. This driver is an adaption of the generic UART driver to the special requirements of the BLACKfin due to the dynamic system clock selections. In contrast to providing fixed tables with the serial dividers required by the generic driver the BLACKfin clone calculates the divider based on the system clock speed and the speed of the serial port selected during the configuration of the eCos system. The BLACKfin port uses a function rather than a fixed preprocessor macro or table based approach to support further planned enhancements to eCos providing extended dynamic power management support. As the power management on the BLACKfin can effect the core and system speeds either separately or collectively the divider value might be affected by a change of the power settings. The fixed table based approach of the original serial driver or even a preprocessor directive implementation would not cater for this case and would require further adaption of the driver.
In addition to the dynamic power management the HAL also profits for different variants and platforms being able just to provide the system clock multiplier ratios and the input frequency together with the serial port speed rather than having to provide multiple tables with calculated values.

6.1.4. Configuration options

Most configuration options have been placed in the appropriate HAL. Each part of the HAL will provide extensive options that influence the system. While the variant HAL only supports the selection of the processor type the platform HAL will determine the processor input clock, the memory layout and startup types and the serial port speed for diagnostic output. The architecture HAL will provide most options even if some are influenced by the variant and platform selections. This approach has been chosen to have a common place for the configuration options covering the same topic and parts of the architecture rather than having to set the options in every part of the HAL.
One example are the system clock speed settings that have been placed in the architecture HAL as the settings and options are architecture specific but influenced by the settings provided by the platform and variant HAL. The selection in the variant HAL will influence the settings of the platform HAL which will itself alter the architecture settings. The architecture is however the place where these settings belong based on the hardware structure.

6.1.5. Driver support

The thesis does further provide upper layer support for devices that are not required for the HAL. These devices include the serial port and the ethernet on the STAMP BF-533. The serial driver and the ethernet both support polling and interrupt driver interaction. Both drivers are based on generic drivers. The serial driver has been adapted for the specific requirements of the BLACKfin serial controller.
The ethernet driver has been adapted to meet the special implementation of the STAMP platform that uses memory sharing between the ethernet controller and the asynchronous memory banks. No driver support has been implemented for the SPI, PPI and SPORT devices on the BLACKfin as the stable version of eCos does not provide support for these device classes.

6.2. Portability of the current HAL

The current implementation extends the portability of eCos by providing support for the BLACKfin architecture. In order to test the portability the standard eCos tests have been run on the architecture. In addition the portability to a slightly different platform and variant have been tested during this thesis to estimate the amount of work required for a variant and platform port.
The second platform chosen for testing was the STAMP BF-537 board.
In order to provide basic support for the processor only the cache sizes had to be adapted. These sizes might have influences on programs that require the exact amount of cache available to optimize the data and instruction placement in memory. In addition the changed mapping of the interrupts on the system interrupt controller and the extension from 24 to 32 system interrupts has been incorporated. Furthermore, the platform required adjustment.
The changes to the platform HAL itself are marginal. The only changes necessary were the input clock frequency for the processor and the memory layout due to a smaller synchronous memory on the board.
In addition the basic portability test required changes to the driver inline codes that are provided. This code describes the amount of devices supported by the same driver that are available in the system and their specific configuration settings. In case of the BF-537 the amount of serial ports increased from one to two. In addition to this the system interrupts used by both serial ports had to be adapted as the BF-537 uses a different interrupt mapping on the system interrupt controller. The driver for the ethernet chip SMSC 91c111 could not be used as the BF-537 provides an inbuilt ethernet MAC controller that is not compatible to the SMSC 91c111. With these changed settings and an adaption of the clock multiplies to operate the BF-537 within its specified operating conditions it was possible to boot the STAMP BF-537 board. The test provided by eCos did pass as they did on the STAMP BF-533. It was further possible to run the eCos based boot manager RedBoot on the STAMP BF-537. But due to the lack in support for the ethernet MAC no network connection could be achieved. This behavior was expected.

6.3. Limitations of the current port of the BLACKfin architecture

The current implementation of the BLACKfin architecture is limited in its direct support for the inbuilt devices to the serial connection and the core timer and interrupt and exception system. In addition the watchdog and the real-time clock are supported by this port.
The STAMP-BF533 is in addition to this supported with its memory layout and ethernet chip.
The current port does not provide support for driver classes that are currently not supported by the stable eCos branch. This includes the classes of PPI, SPI, SPORT, timer and CAN devices. SPI and CAN support are included in the current development release of eCos.
In addition to the limitation of driver classes the drivers implemented do not support direct memory access or DMA transfers. The DMA transfer could be used for the serial ports on the BLACKfin. In case DMA is used the underlying caching strategy has to be altered to support areas that are not cached in order for the processor to notice changes made by the DMA system in memory.
While the HAL does fully support write-through mode caching with and without allocate on write with all page sizes the support for write-back is limited.
It is unsafe to disable the data cache when operating in write-back mode. This is due to the limited flushing capabilities of the data cache. It is not possible to flush an entire page or the entire cache with a single instruction.
Flushing is either done by directly flushing the cache line in question. For page sizes of 1 kB and 4 kB this can be done via a loop iterating all possible locations.
The BLACKfin uses a 32 byte cache line size which yields a maximum of 128 cache lines for a 4 kB cache. A 1 MB page however would require 32,768 possible cache lines to be tested by iteration which is more than the maximum of 1024 lines available on the BF-533.
The only way of efficiently implementing the flushing for pages with a minimum size of 1 MB or the entire cache would be to iterate through every cache line in the cache via the data cache testing registers of the BLACKfin. In case a valid and dirty cache line is found that matches the page in question this line would have to be cached.

6.4. Verification

In addition to intensive testing parts of the code have been considered for verification.
The verification was done by examining the code, its pre- and postconditions.
It was checked whether all conditions were met by mathematical means.
This included testing for proper declaration of registers used and clobbered. In addition it was tested whether all the registers used were either declared to be changed or had been saved and restored within the process of executing the code to be verified.
The verification was performed for the interrupt and exception trampolines, default exception VSR and the default interrupt VSR. It was further checked whether the context switching code does properly address all cases of system states described in the design section and whether it will store and restore all registers as defined.
All parts of the code tested did pass the verification.

6.5. Testing

The implementation of the HAL and drivers has been tested with the test programs provided by eCos. And a limited set of additional tests. The additional tests included the testing of the interrupt global enable /disable macros in combination with the mask and unmask macros for interrupts. The special test was required as this is an architecture dependent implementation problem that is not covered by the standard tests.
The HAL did pass the tests and globally enables and disables interrupts while still performing masking and unmasking operations without overriding the global interrupt status.
While performing the general function tests the HAL did pass the tests supplied for the provided drivers and HAL, kernel and library functions. This does however not guarantee the correctness of the code. It was further found that some tests provided by eCos do not perform sufficient testing to actually decide whether a given piece of code passes or fails. One example is the HAL test for context switching. The first basic implementation did pass the test even though it should have failed. The first implementation step did only cover the two compatible cases. The context switching test did only test one of those. It did not test whether context switching is possible for threads being in incompatible states.
It was further found that the mutex3 and kmutex3 tests for testing the proper implementation of the mutex synchronization primitive of the kernel do test all cases of context switching. The basic implementation lacking support for the incompatible states did fail this test as expected. The final complete implementation of the design which does adjust incompatible states passes the mutex test.
Based on these findings special considerations had to be given to the testing code itself for correctness and completeness.
Serial and ethernet device drivers were tested with both non-interrupt driven or polling mode and interrupt driven mode and passed both.
An additional testing focus was based on caching. The HAL does properly handle the miss exceptions and replaces page table entries if required. It observes the configuration settings to determine the page options. In addition it has been tested if the HAL will ensure its own pages are always locked and never replaced if caching is enabled. The HAL did pass this test as no pages got replaced. The HAL did also pass the test for all pages being locked. It reacted with a HAL panic halting the system as expected.
The last focus for testing was based on a complex example application that was provided by eCos itself, RedBoot. The boot manager is a complex application that provides support for downloading images via serial line and ethernet. It was tested in RAM startup as well as in ROM startup. It was used to test the proper initialization of system devices on startup including power management setting. RedBoot did perform as expected in all situations.

6.6. Measurements

eCos provides a test program for testing important timing parameters of the operating system. The tm_basic test program will test system functions for thread management of the kernel like creating and destroying threads. The program will also test the performance of synchronization objects provided by the kernel like mbox, mutex and semaphores. Further tests include the processing time of the core timer interrupt, alarms and counters.

data	*off*	*1 kB*	*4 kB*	*1 MB*	*4 MB*
create thread, wt	14.28	28.20	8.55	5.49	5.43
create thread, wb	14.28	33.87	9.85	5.24	5.21
clock-int, wt	16.97	2.86	2.87	2.88	2.86
clock-int, wb	16.97	1.88	1.88	1.87	1.86
Tick & fire counters [>1 together], wt	190.13	41.42	41.43	41.41	41.42
Tick & fire counters [>1 together], wb	190.13	21.07	21.08	21.08	21.07

Table 3: timing of system tasks in µs, wb = write-back, wt = write-through

Table 3 shows some basic timing aspects of the system for different caching strategy and page sizes. The diagram shown in 16 is a graphic representation of the data shown in the table.
The table shows a visible influence of the data cache on the timing of system tasks.
All measurements were taken with enabled instruction cache and 4 MB pages.
The data demonstrates the influence of the caching strategy and page sizes on the performance of the system. It allows shows that not all tasks are affected by differing cache sizes.
An increased page size does generally result in a better performance. It can also be seen that write-back is usually better than write-through.

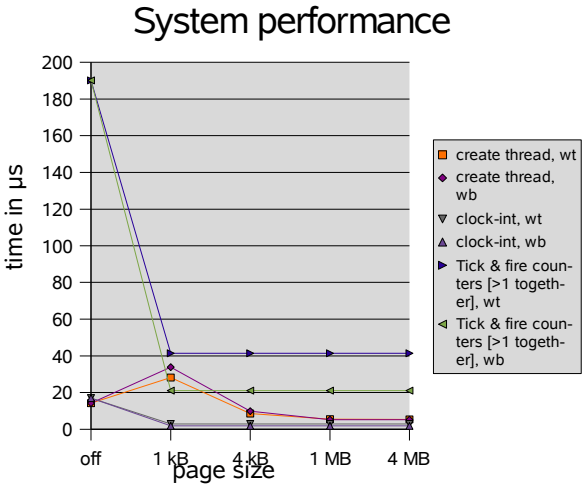


Illustration 16: performance of selected tasks

The values for creating new threads in the system show an anomaly though.
The system takes longer using write-back with small page sizes than write-through even though it is faster than write-through with page size greater than 4 kB.
This is due to the overhead associated with page table entry replacements in relation to the page size. While replacing pages in write-through mode does not require synchronization the replacement of a write-back page will result in a flush of the page. This means that the data is now written back to memory. This is especially noticeable due to the small size of the page and the limit to 16 entries in the page table.

These factors lead to frequent page replacements due the use of memory. The overhead associated with flushing the pages has a big influence on the performance.
The overhead for frequent page replacements of small pages is also responsible for the increase of the time for creating a task when using caches with 1 kB size compared to not using the data cache.
The instruction cache which had been enabled for all the data shown in the table also has a substantial influence on the system. The influence is clearly illustrated by comparing the values for data cache disabled and instruction cache enabled to both caches disabled.

	DCACHE disabled ICACHE disabled	***DCACHE enabled (4 MB) ICACHE disabled***	***DCACHE disabled ICACHE enabled (4 MB)***
create thread	44.71	31.42	14.28
clock-int	37.84	20.40	16.97
Tick & fire counters	409.16	227.44	190.13

Table 4: performance influence of the ICACHE and DCACHE, times in µs

The results are shown in table 4. It is clearly visible that the time with instruction cache disabled is by the factor two to three longer than with instruction cache.
Also visible from the table is that the impact of the instruction cache on the system performance for the test cases of tm_basic is larger than the impact of the data cache.

7. Future work

The result of this thesis can be used as the basis of a range of different further work.
In addition to a transfer from the experimental phase to a working environment extended support for device drivers could be included. This can be based on the current development branch of eCos to support drivers for the controller area network CAN on the BF-537 and the SPI for all BLACKfin processors. In addition a driver for the ethernet MAC on the BF-537 would provide enhanced support for the new generation of STAMP boards based on this processor type. This would allow the board to be used for teaching purposes as all basic devices would be support by eCos on either version of the STAMP board.
Future work can also be done to further improve the features of the BLACKfin HAL rather than adding support for new platforms.
An important enhancement would be a fully functional implementation for the synchronization macros of eCos to support write-back with all page sizes.
This in turn could be the basis for extensive study of the influence of caching on the performance of programs in different environments. These environments can either be determined by different page sizes for either type of cache but also by the strategies used to place data and code into memory and their influence on the cache performance.
Another area of interest are the security aspects of the MMU attached to the pages.
An investigation into applications that could profit from the security features following different strategies of providing read and write functions to pages and strategies for dealing with protection errors would be just one field of interest. It could be investigated whether a protection of for instance stack boundaries is feasible with the current structure of eCos in which the HAL has no knowledge about the location of stacks in memory.
Connected with this would also be the question if the introduced overhead for any of the suggested features is justified by the benefits of them.
Also connected to the MMU with regard to caching and security would be the question whether a mixture of different page sizes or a single page size per cache offers better performance and more flexibility. The flexibility however could lead to problems like multiple page entries in the MMU referring to the same memory location. This case would allow studies of strategies dealing with these situations.
Should these situations be avoided due to increased overhead and how big is the influence on the overall system performance.
Should the least or most restrictive page setting be applied and whether security, caching or both options should be considered.
And how would different page size help to guard the stack by placing smaller pages as guards inside large pages containing the stack.
Further studies could be conducted in the area of page table entry replacements.
Does a more complicated victimization strategy provide better performance despite a possibly increased overhead.
In addition it would be interesting to investigate the relation between regions of RAM that require different page options to having a handler creating the pages on the fly or having them pre-generated in a page table in memory.
This leads to the general question of the influence of page tables. Due to the hardware architecture it would also be possible to examine the different approaches of traversing and structuring page tables in memory on the performance of the system.
This is possible because the BLACKfin architecture does not have a fixed format of the page table and does not require a table based approach to be used.

8. Conclusion

In this thesis I have shown the steps involved in porting eCos to a new architecture including the different design aspects. After providing an insight in the different design options the thesis did also provide detailed implementation information for selected cases that posed a particular problem on the BLACKfin architecture. This thesis has shown in particular how interface definitions abstracting from the underlying hardware to provide a common view across different hardware platforms can prove to be difficult to be implemented on a particular platform and how these problems have been solved for the BLACKfin. I further detailed the evaluation of the current state of the porting process and the testing, verification and timing process involved. This thesis finished by providing aspects that could provide starting points for future work based on the BLACKfin architecture.

9. Appendix

9.1. Illustration Index

9.2. Code Index

9.3. Table Index

9.4. References

ANALOG 2006a Blackfin Processor Core Basics
http://www.analog.com/processors/blackfin/overview/blackfinCoreBasics.html
last accessed : 27.09.2006

ANALOG 2006b ADSP-BF533 - High Performance General Purpose Blackfin Processor
http://www.analog.com/en/prod/0,2877,ADSP%252DBF533,00.html
last accessed : 26/09/2006

ANALOG 2006c BLACKfin Processor Architecture Core
http://www.analog.com/processors/blackfin/images/Blackfin_architecture.gif
last accessed : 28/09/2009

ANALOG 2006d Functional Block Diagram
http://www.analog.com/en/imageDisplay/0,2883,,00.html?
Path=%2Fimages%2FProduct%5FDescriptions
%2F34777385905585200 32BF533%5Flg2
%2Egif&Title=Functional+Block +Diagram
&alt=ADSP-BF533+Blackfin+Processor
last accessed : 28/09/2006

ECOS 2006a eCos(TM) Reference Manual
http://ecos.sourceware.org/docs-latest/pdf/ecos-ref.pdf
last accessed : 05/05/2006

ECOS 2006b eCos User Guide
http://ecos.sourceware.org/docs-latest/user-guide/ecos-user-guide.html
last accessed : 28/09/2006

ECOS 2006c RedBoot User's Guide
http://ecos.sourceware.org/docs-latest/redboot/redboot-guide.html
last accessed : 28/09/2006

ECOS 2006d eCos
http://ecos.sourceware.org
last accessed : 29/09/2006

ECOS 2006e eCos reference guide
http://ecos.sourceware.org/docs-latest/ref/ecos-ref.html
last accessed : 29/09/2006

9.5. Bibliography

ANALOG 2006a Blackfin Processor Core Basics
http://www.analog.com/processors/blackfin/overview/blackfinCoreBasics.html
last accessed : 27.09.2006

ANALOG 2006b ADSP-BF533 - High Performance General Purpose Blackfin Processor
http://www.analog.com/en/prod/0,2877,ADSP%252DBF533,00.html
last accessed : 26/09/2006

ANALOG 2006e Datasheet ADSP-BF533
http://www.analog.com/UploadedFiles/Data_Sheets/320040195ADSP_BF531_2_3_b.pdf
last accessed : 04/05/2006

ANALOG 2006e ADSP-BF533 Hardware Reference Manual
http://www.analog.com/UploadedFiles/Associated_Docs/892485982bf533_hwr.pdf
last accessed : 29/06/2006

ANALOG 2006f ADSP-BF537 Hardware Reference Manual
http://www.analog.com/UploadedFiles/Associated_Docs/4206716165649BF537_HRM_whole_book_o.pdf
last accessed : 29/09/2006

ANALOG 2006g BLACKfin Processor Programming Reference Manual
http://download.analog.com/dsp/manuals/Blackfin_PRM.pdf
last accessed : 03/05/2006

ANALOG 2006h Silicon Anomaly List ADSP-BF533
http://www.analog.com/UploadedFiles/REDESIGN_IC_Anomalies/691920698IC_Anomaly_BF533.pdf
last accessed : 20/09/2006

ANALOG 2006i Datasheet ADSP-BF537
http://www.analog.com/UploadedFiles/Data_Sheets/ADSP-BF534_BF536_BF537.pdf
last accessed : 20/08/2006

BLACKFIN 2006 Selection of Datascheets and schematics of the STAMP board
http://blackfin.uclinux.org/frs/download.php/280/STAMP_Datasheets_v1.0.zip
last accessed : 19/09/2006

ECOS 2006a eCos(TM) Reference Manual
http://ecos.sourceware.org/docs-latest/pdf/ecos-ref.pdf
last accessed : 05/05/2006

ECOS 2006b eCos User Guide
http://ecos.sourceware.org/docs-latest/user-guide/ecos-user-guide.html
last accessed : 28/09/2006

ECOS 2006c RedBoot User's Guide
http://ecos.sourceware.org/docs-latest/redboot/redboot-guide.html
last accessed : 28/09/2006

ECOS 2006d eCos
http://ecos.sourceware.org
last accessed : 29/09/2006

SMSC 2006a SMSC 91C111 Datasheet
http://www.smsc.com/main/datasheets/91c111.pdf
last accessed : 20/06/2006

MASSA 2006 Embedded Software Development with eCos
http://www.informit.com/content/downloads/perens/0130354732.pdf
last accessed : 28/09/2006

UCLINUX 2006 uClinux
http://blackfin.uclinux.org
last accessed : 29/09/2006

9.6. Content of DVD

The DVD included in this thesis contains the code developed and written as part of this work. The documents on the disc are structured as follows.

```
root
|
|-archive
|       |-ecos.tar.bz2          (entire eCos 2.0 stable archive including the bfin HAL)
|
|-code                          (subdirectory containing the source files as found in eCos)
|
|-doc
|       |-thesis.pdf            (this document in electronic form)
|
|-tests                         (containing the output of test programs)
```