Porting the Mutek operating system to ARM platforms

Nicolas Fournel, Antoine Fraboulet, Paul Feautrier

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

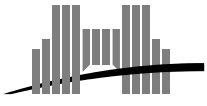
Porting the Mutek Operating System to ARM platforms

Nicolas Fournel
Antoine Fraboulet
Paul Feautrier

March 2006

Research Report N° RR2006-12

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr



Porting the Mutek Operating System to ARM platforms

Nicolas Fournel
Antoine Fraboulet
Paul Feautrier

March 2006

Abstract

This report presents the work done on modifying the lightweight Mutek operating system to add support for two complex Arm-based SoC architectures. Both of these platform use nearly the same ARMv4 core CPU model but have a different memory map and integrate different system peripherals such as interrupt controller, timer and serial interfaces. An initial support for MMU operations using an identity mapping has also been added to the hardware abstraction layer. This support was compulsory to access hardware information needed to activate the data cache.

Keywords: Mutek, Operating System, ARM Platform, Porting

Résumé

Ce rapport de recherche présente les modifications faites sur le système d'exploitation léger Mutek pour ajouter le support de deux plateformes complexes utilisant des processeurs ARM. Les deux plateformes possèdent une version ARMv4 du cœur de processeur mais utilisent des adressages mémoires ainsi que des périphériques différents pour le contrôle des interruptions, les timers et les ports de communication série. Un support préliminaire de gestion de la mémoire virtuelle utilisant la MMU a projection identité a également été rajouté dans la couche d'abstraction matérielle. Ce support est nécessaire pour ajouter les fonctions d'activation des caches de données des processeurs.

Mots-clés: Mutek, Systèmes d'exploitation, Plate-forme ARM, Adaptation

Contents

List of Figures

1 Introduction

The Mutek operating system is available as part of the DISYDENT *Open Embedded System Development Environment* [4, 6]. The Mutek operating system is a lightweight kernel that proposes an implementation of the POSIX threads for multiprocessor platforms with shared memory on multiple memory banks.

Mutek has been developed originally for MIPS R3000 processors. It has then been ported to SPARC V8 and preliminary support for ARM and POWERPC is currently available within the CVS repository. So far the Mutek kernel has only been used, to our knowledge, within the DISYDENT framework and with the SystemC simulation platform available from the SOCLIB initiative [1]. These simulation environments propose design models and tools that can be used for hardware/software codesign of complex System on Chip (SoC). These models include processors, system interconnect buses and Network on Chip, RAM modules and some other system peripherals. However the booting process and input/output peripherals available as part of the design libraries are not as complex as real peripherals.

As part of our work on embedded systems we need a very lightweight operating system to build power measurement benchmarks. This research report presents the modifications made to the Mutek kernel in order to boot it on the ARM Integrator CM922T-XA10 platform [2] and on the skyeye [9] system simulator for the Atmel AT91RM200 SoC. The interested reader should note that this report is related to the Research Report 2006-08 [5] on porting the Linux and uCLinux kernel on the same platform.

The next section presents the original software architecture of the Mutek operating system and its associated libraries.

1.1 The Mutek Kernel

The Mutek kernel uses a monolithic architecture in which both the operating system and application code are statically linked at compile time. The Mutek source code can be parameterized in a number of ways through the use of pre-processor variable definitions within the code. Once the libraries have been compiled for a specific target architecture the application code can be linked to the software libraries in order to make an ELF file that can be used to boot the platform and run the application code. Figure 1 presents the basic operations to build a self-contained ELF file. Mutek uses a flat memory model running in physical address mode. The application code is run within the same memory address space as the kernel services thus allowing a complete control over the memory allocation of the objects in exchange for the memory protection mechanism that are usually available on more complex systems. The linker script configuration file (`ldscript`) must be explicitly given during the linking process. This file defines the memory mapping of all objects and sections used within the compiled code.

The Mutek functionalities are separated into the following libraries

- `libhandler`: platform specific code. The library proposes a hardware abstraction layer on top of which all the Mutek functionalities are built. This library contains the platform specific assembly source code.
- `libpthread`: Posix thread implementation that conforms to the Posix 1003.1 standard.
- `libc`: tiny libc library that can be used by application code.
- `libdpn`: Disydent Process Network library. This library proposes communication channel abstraction that can be used to hide hardware and software communications behind a unified fifo-based communication framework.

In this report we will only focus on the `libhandler` library and we propose some extension for the input output mechanisms contained in the `libc` library. The `libpthread` was portable enough to remain unchanged while being used on another architecture than the one it was primarily designed for. We also did not modify the `libdpn` library.

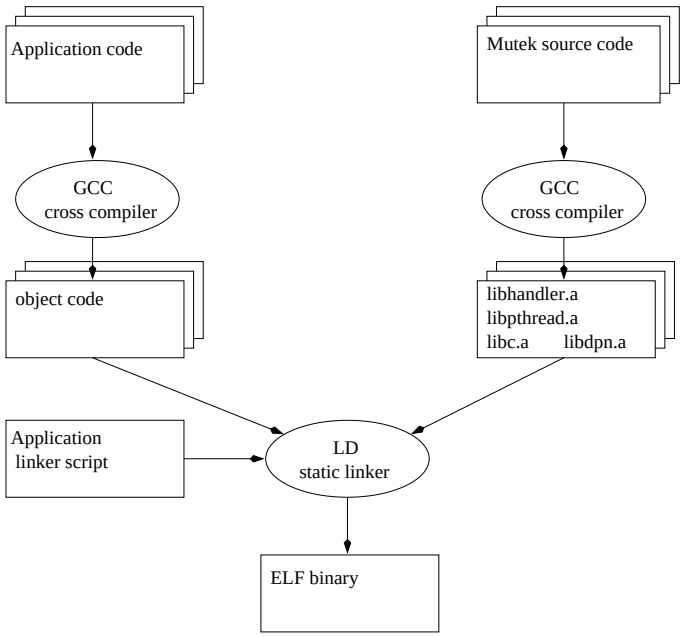


Figure 1: Basic steps to compile Mutek source and application code

1.2 The `libhandler` library

The `libhandler` library is used to build a hardware abstraction layer. This code library regroups hardware dependant part of some common functions used within the Posix thread API implementation. Amongst other

- Hardware specific type definitions and macros
- Processor bootstrap and reset code
- Software context switch
- Mutual exclusion mechanism implemented using spin-locks
- Interrupt controller
- Timer interface definition

The current Mutek implementation can only use the peripherals available within the DISYDENT and SOCLIB environment. These environments include an interrupt controller, a timer and a basic serial character output device. CPU related functions for ARM based SoC are presented in section 2.1.1. Memory mapping and devices for our platforms are presented in sections 2.2 and 2.3.

1.3 The `libc` library

The Mutek libc is used by both the implementation of the pthread library and can be used by the application code. The functionalities included in the `libc` library include basic C support functions:

- `malloc`: Memory allocation for both global and local storage. These memory allocation functions can be used on an architecture with explicit multi-bank memory regions.
- `ctype`: character type handling.
- `string`: C string support and memory manipulation functions.

- `stdlib`: standard C library functions (`atoi()`, `bsearch()`, `exit()`, `random()`, `strtol()`).
- `stdio`: basic output text console handling.

The current version of the Mutek libc was left untouched except for the text output functions such as `printf()`. Both DISYDENT and SOCLIB support a very simple `TTY` output device that can take characters on a single memory mapped register (`__tty_addr`). The pseudo device is used within simulations to have a direct feedback from the application code using a terminal type emulator. The modifications we made in order to support a console interface that can be used over a serial port are presented in the device driver section 3.2.

1.4 Porting Mutek to a real platform

This report presents the work done on modifying the Mutek sources to add support for two complex ARM-based SoC architectures. Both of these platform use nearly the same ARMv4 core CPU model but have a different memory map and integrate different system peripherals such as interrupt controller, timer and serial interfaces. An initial support for MMU (Memory Managment Unit) operations using an identity mapping has also been added to the hardware abstraction layer. This support was compulsory to access hardware information needed to activate the data cache.

2 Platforms Description

2.1 ARM Common Architecture

The two platforms are based on an ARM processor. For the AT91RM9200 SoC it is an ARM 920T and for EPXA SoC it is an ARM 922T. First of all, it is interresting to note that ARM 920T and ARM 922T are nearly the same processor. In fact the only difference between the two of them is their cache sizes. ARM 920T owns 16kB of instruction cache and 16kB of data cache, whereas ARM 922T only gets 8kB for instruction cache and 8kB for data cache. The remainder of the two processors is strictly identical. They have the same core, ARM9TDMI and the same MMU, ARMv4 MMU. The global architecture of the CPU is thus the same for the two platforms and is depicted in figure 2.

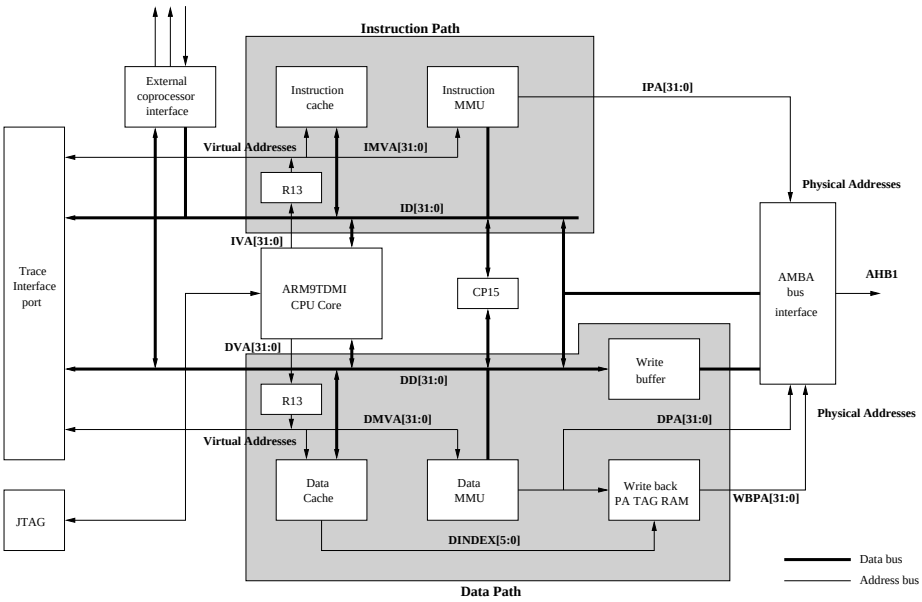


Figure 2: ARM 920T/922T architecture

2.1.1 ARM9TDMI Core

We will give in this section more details about the ARM9TDMI core included in our two target CPUs.

Instruction Set Architecture (ISA): ARM9TDMI is a RISC CPU core, whose pipeline is divided into five stages : Fetch, Decode, Execute and two Load and Store stages. This core fully implements the ARM 32 bit instruction set until revision 4. It also implements the reduced 16 bits instruction set called Thumb. More recent ARM technologies like DSP instructions or Java accelerations instructions (Jazelle) are not supported by this core.

Execution Modes: ARM processors have different execution modes. There are 6 modes in older processors (ARMv3, and before) : User, Supervisor, Abort, Undefined, IRQ and FIQ modes. In more recent processors (ARMv4 and above) a seventh mode appeared called System mode. Among these modes five are exception modes and only one is an unprivileged mode. Indeed, User mode is the normal execution mode, since it is unprivileged, and Supervisor, Abort, Undefined, IRQ and FIQ are exception mode. The System mode is the only privileged mode which is not an exception execution mode.

Exceptions: As we just said, there are five exception execution modes, but this does not mean that the processor has only five exception types. In fact, there are 6 different exceptions : Reset, Abort, Undefined, SWI, IRQ and FIQ. Here are more details about these exceptions.

Reset is raised when the reset input of the processor is asserted. Abort regroup instruction fetch abort and data access memory abort, which means that an error occured while accessing the memory for instruction reading or data access. Undefined exception occurs when an unknown instruction is executed on the CPU. The three remaining types are interruption exceptions. The first, SWI, represents software interruptions, the second, IRQ, normal hardware interruptions and the third, FIQ, fast hardware interruptions.

For most of the exception types, the mapping of their execution mode is obvious. For the remainder, Reset and SWI are executed in Supervisor mode.

Let us give a few details about how the CPU behave when an exception raises. On exception assertion, the instruction execution is stopped and the processor jumps to the CPU Vector regrouping all software exception handlers. To do so, the program counter is updated with a value obtained by adding the CPU vector base (`0x00000000` or `0xFFFF0000` depending on the CPU configuration) and an offset dependent on the type of exception. For exemple if the CPU vector base is `0x00000000` and an IRQ occured, the CPU program counter jumps to the address `0x00000018`. At this address we generally find a jump instruction (`b` branch) to the real handler address.

Register file: The final point of the core description is the register file. ARM CPUs have a 16 general purpose register file, `r0` to `r15`. An extra register `CPSR` (Current Processor Status Register) is available, it represents the current processor status. Among the 16 registers mentioned before, one is the program counter (`PC`), `r15`. Two other registers are used for special purposes, `r13` as stack pointer (`SP`), and `r14` as link return (`LR`). Which means that when executing a `bl` (branch and link) instruction, the return address (address of the `bl` plus 4) is stored in `LR=r14`.

In fact ARM CPUs do not have only 16 general purpose registers and one status register, but 32 general purpose registers and 6 status registers. On top of these 17 registers (`r0` to `r15` and `CPSR`), there are banked registers. The banked registers are available in exception execution modes. For general purpose ones,they are available replace the normal registers in the 16 registers scope of the CPU. Banked status registers give access to an extra register, the saved CPU status register (`SPSR`). Figure 3 shows all banked registers in a gray filled box. For example, when running in IRQ mode, the `r13=SP` and `r14=LR` registers available are not `r13` and `r14` of the User/System mode, but `r13_irq` and `r14_irq`.

On mode switching, from user (or exception) to exception mode, the `CPSR` is saved in the `SPSR` of the target execution mode, as well as the `PC` is copied in the `LR` of the exception mode.

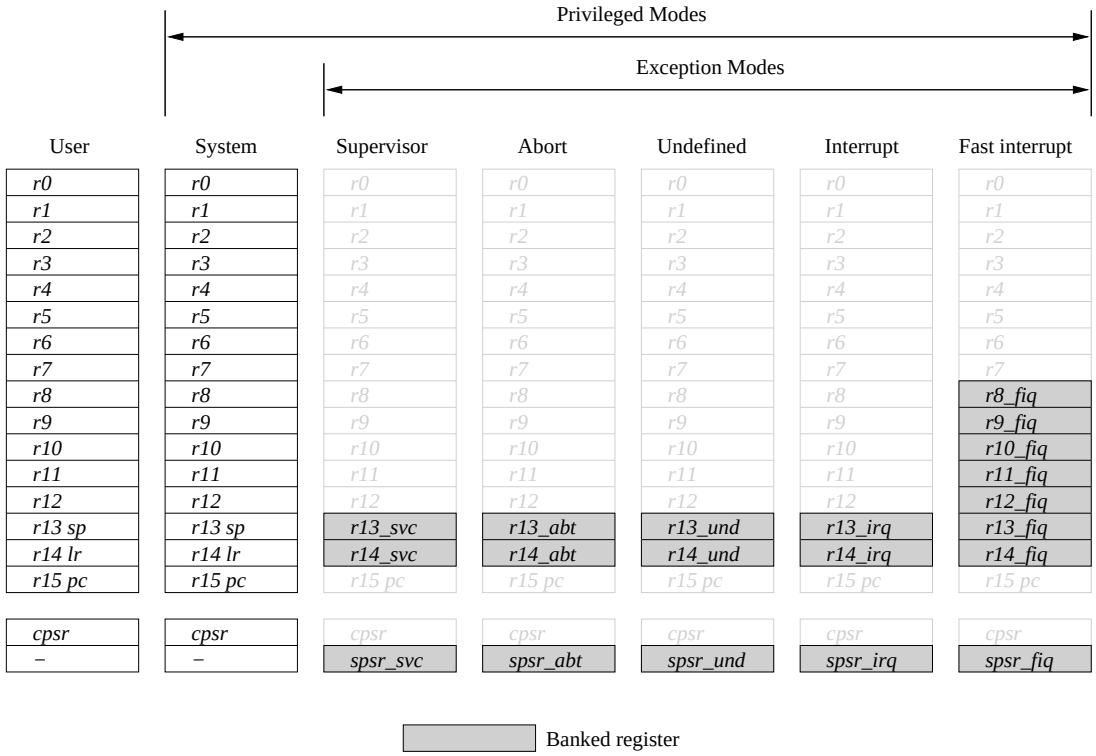


Figure 3: Banked registers of ARM CPU

2.1.2 Instruction and Data Caches

The ARM 920T and the ARM 922T have seperated instruction and data caches. Their size is 16 kB in the 920T and 8kB in the 922T. As far as the write buffer is concerned, its size is 16 words in the two processors.

These caches are virtually addressed, which means that cache lines are indexed by virtual addresses. On top of that, the data cache can only be used when MMU is activated. In fact the control bits, which indicates if the cache and the write buffer must be used for a memory region, are part of the page table entries. Between the two processors, only one parameter changes, the size of the sets. In the ARM 920T they are 128 lines deep and in the ARM 922T they are 64 lines deep. Thus, the 920T caches are made of 512 lines and the ARM 922T caches are made of 265 lines. The line size is 8 words (32 bytes) for the two of them and they are organized in a 4-way associative way.

2.1.3 MMU Behaviour

The MMU integrated in the ARM 920T and in the ARM 922T are the same, ARMv4 MMU. We will give here more details about the behaviour of this MMU.

First of all, we must underline the fact that instruction and data have their own TLB (Translation Look-aside Buffer), translation cache. These TLBs have a 64 entry width.

In the ARMv4 MMU, memory can be accessed through four different page or section sizes : 1kB (tiny page), 4kB (small page), 64kB (large page), 1 MB (section). These pages and sections are accessible through one or two stage table walking depending on page-mapped or section-mapped access.

Translation base: the translation starts when TLB contains no translation for a virtual memory address. Then the translation table base (TTB) address gives the location in physical memory of the first stage table. This register is in the configuration coprocessor CP15.

Level 1: Translation Table. The first stage table is also called translation table. It is 4096 entries long, which means that its size is 16 kB. Each entry in this table represents 1MB of virtual memory. There are four different entry types in this table as can be seen on Figure 4.

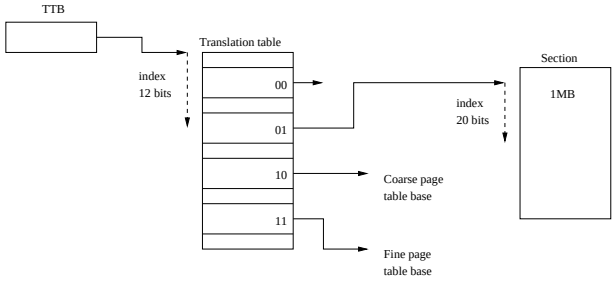


Figure 4: ARMv4 MMU level 1

The first type is the section descriptor. As we said before, memory is accessible through section. Obviously these sections have a size of 1 MB. All informations about domain protection, cache and write buffer are present in this first level entry.

The two following types of entries are page entries. There are two different types of page table : coarse grained and fine grained. The first level entry in these case only gives the base address of the corresponding second level page.

The fourth type is undefined and generates an error.

Level 2: Page tables. At this level, we have two different types of page table, the coarse grained and the fine grained page table. As their name indicates, the first type describes its 1Mo region with big pages and the second one with smaller pages.

The page size can have three different sizes : 1kB (tiny page), 4kB (small page), 64kB (large page). Coarse grained page table entries can only describe small or large page. However, Fine grained page table entries can describe tiny, small or large page entries. This difference is due to the following fact: coarse grained page table divides its 1MB in 4kB blocks, so it has only 256 entries. When a large page descriptor is set, it is repeated on the 16 contiguous blocks descriptions. For fine grained page table the situation is the same, but it divides its 1MB in 1kB blocks (it is 1024 entries). If a small or a large page description is set the description is repeated on all the blocks descriptor the page contains. The figure 5 summarize this second level description relations.

To conclude, figure 6 give an global view of the two level page table organisation.

2.2 Integrator CM922T-XA10 Platform

We will have a closer look on the pecularities of the two platforms presented before. The first platform to be detailed is the ARM Integrator/CM 922T-XA10 in standalone mode. This platform is based on an Altera Excalibur EPXA10. In the remainder of this section we will only talk about EPXA10 hardware details, since Integrator/CM has only few hardware embedded on board. A more detailled description of the platform is available in [5], the following parts will only underline major aspects.

2.2.1 Hardware Architecture

Before going any further, here is a global overview of the hardware architecture of the EPXA10 as it is integrated on the CM 922T-XA10. The figure 7 depictes this architecture. CM 922T-XA10 specific hardware is implemented in the FPGA of the EPXA10.

This SoC (System-On-Chip) contains all required hardware to make an Operating System (OS) boot on it. Indeed, hardware pieces necessary for us are serial port (UART), timer and interrupt controller. To conclude, it is interesting to underline that we have different types of memory available on the CM 922T-XA10. Main memory (DRAM) is embedded on the board (128 MB). We also have static RAMs usable

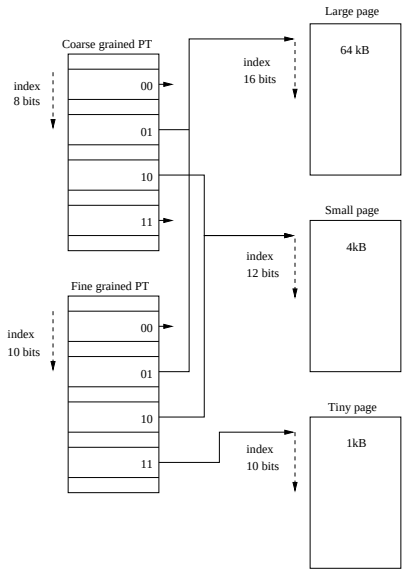


Figure 5: ARMv4 MMU level 2

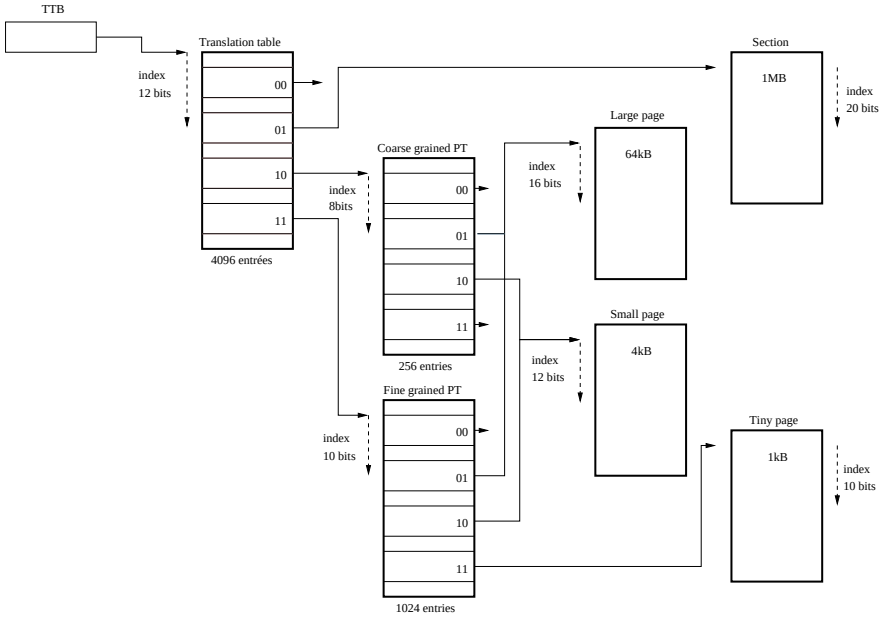


Figure 6: Global MMU access diagramm

as Scratch PADs. This static RAMs are Single port and Dual port static RAMS (SP / DP SRAM) and are integrated in the EPXA10. On top of that, CM 922T-XA10 board also integrates an SSRAM chip (2MB).

2.2.2 Memory map

All stripe peripherals are controlled by registers mapped in memory. These registers are regrouped in a register bank, the stripe register bank. The default mapping address of this bank is `0x0B000000`. Another control and status register bank corresponding to the CM peripherals is mapped at `0x10000000`.

The remaining of the memory map contains memories mapping (Flash, SDRAM, SP SRAM and DP SRAM). Figure 8 gives a full overview of this memory map.

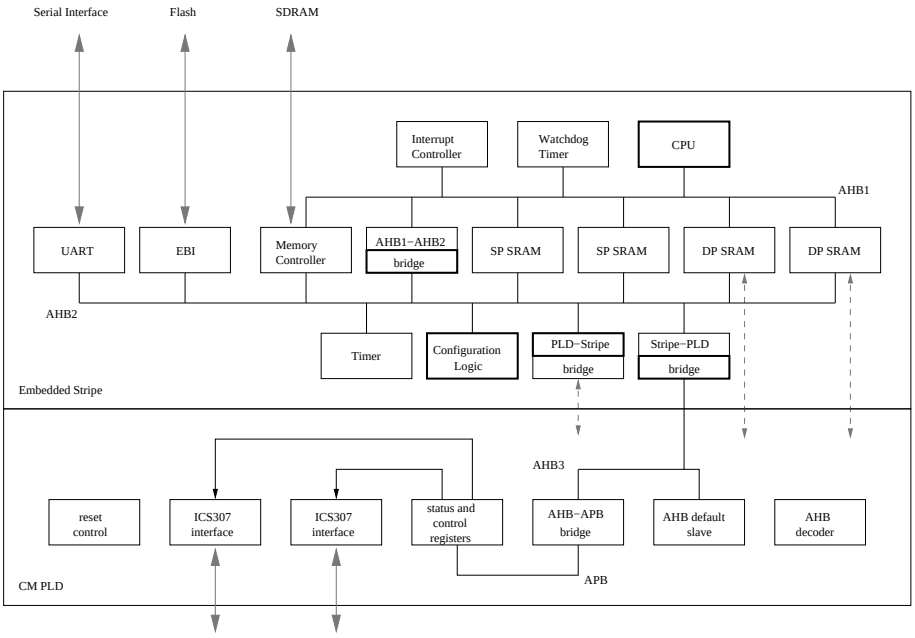


Figure 7: CM 922T-XA10 architecture. Bold blocks are bus masters

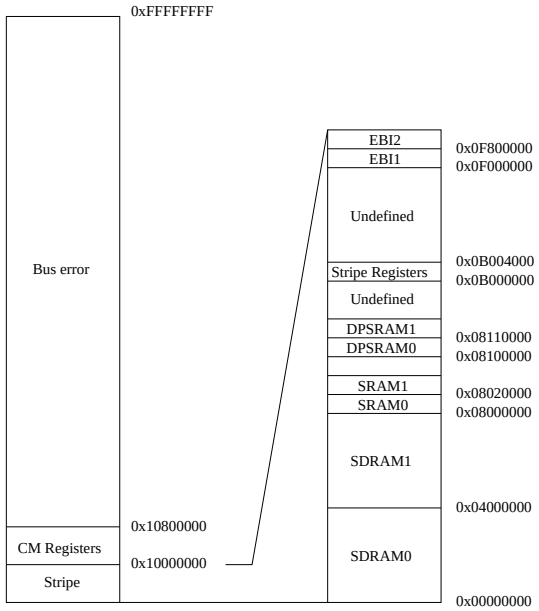
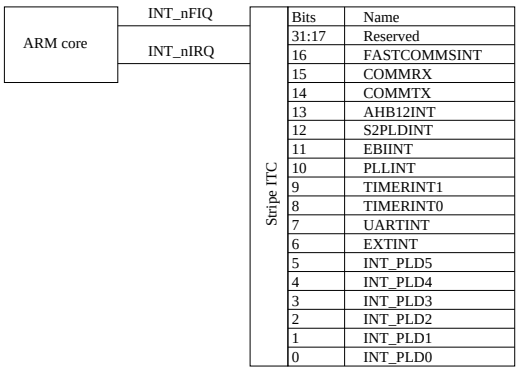


Figure 8: CM 922T-XA10 memory map

2.2.3 Interruption Architecture

EPXA10 embed an interruption controller (ITC) linked to the interruption wires (INT_nIRQ and INT_nFIQ). This ITC gathers all interruptions requests from all hardware of the platform and raises an IRQ or a FIQ on the CPU. All sources are linked to the ITC by only one wire, then the ITC must know if it must raise an IRQ or a FIQ. It has interruption priorities for each source, and decide regarding to this level. The highest level (`0x3F`) corresponds to fast interruptions.



Bits	Name
31:17	Reserved
16	FASTCOMMSINT
15	COMMRX
14	COMMTX
13	AHB12INT
12	S2PLDINT
11	EBIINT
10	PLLINT
9	TIMERINT1
8	TIMERINT0
7	UARTINT
6	EXTINT
5	INT_PLD5
4	INT_PLD4
3	INT_PLD3
2	INT_PLD2
1	INT_PLD1
0	INT_PLD0

Figure 9: CM interrupt architecture

2.3 Atmel AT91RM9200 Platform

The second platform is an Atmel AT91RM9200. This is a SoC (System-On-Chip) used in many embedded commercial products such as PDA (Personal Digital Assistant).

2.3.1 Hardware Architecture

The figure 10 gives a rapid overview of the global hardware architecture of the AT91RM9200. As you can realize the main difference between this architecture and the CM922T-XA10 one, is the fact that the CPU is connected to an AMBA System Bus (ASB) and that the peripherals are only accessible through a bridge (also connected to the ASB bus), since they are connected by an AMBA Peripheral Bus (APB).

Main Memory (SDRAM) is linked to the SDRAM controller in the Expansion Bus Interface (EBI).

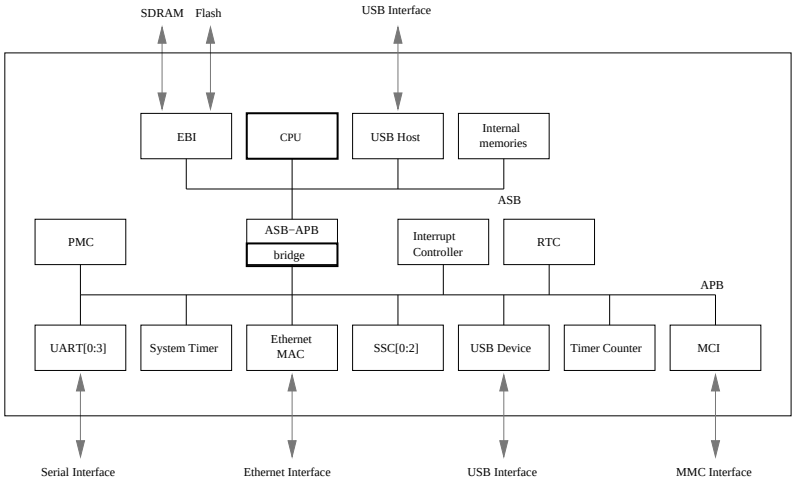


Figure 10: AT91RM9200 architecture. Bold blocks are bus masters

2.3.2 Memory Map

As for CM 922T-XA10, all peripherals can be controlled thanks to memory mapped registers. These registers are regrouped in the last 256 MB of the address space.

The remainder of the physical address space is assigned to memories or left undefined.

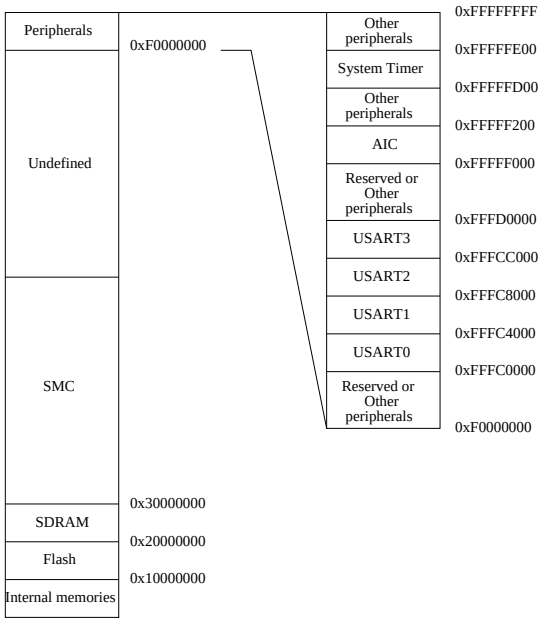
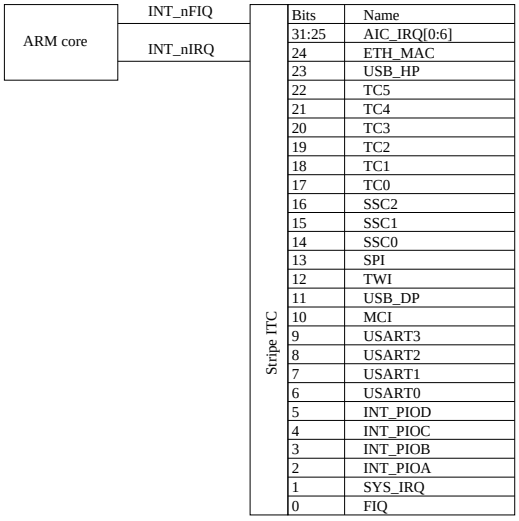


Figure 11: AT91 memory map

2.3.3 Interruption architecture

AT91RM9200 also integrates an interruption controller (ITC). Its behavior and integration is very close to the EPXA10 one. In fact it is linked to the interruption exception wires (INT_nIRQ and INT_nFIQ). The main difference is that interruption source of the ITC can gather multiple hardware sources. For example, source 1 is System interruptions, it regroups System timer interruption, real time clock and so on. On top of that, FIQ managment is completely different since all sources of this ITC raises IRQ except one, source 0, which is specialized to raise FIQ.



Bits	Name
31:25	AIC_IRQ[0:6]
24	ETH_MAC
23	USB_HP
22	TC5
21	TC4
20	TC3
19	TC2
18	TC1
17	TC0
16	SSC2
15	SSC1
14	SSC0
13	SPI
12	TWI
11	USB_DP
10	MCI
9	USART3
8	USART2
7	USART1
6	USART0
5	INT_PIOD
4	INT_PIOC
3	INT_PIOB
2	INT_PIOA
1	SYS_IRQ
0	FIQ

Figure 12: AT91 interruption architecture

3 Mutek Modifications

Porting Mutek to these two real platforms needs to be made in two different phases. The first phase is to ensure that CPU specific routines are available and works fine, and second phase is to port hardware specific drivers.

3.1 Mutek on ARM (generic implementations)

To make Mutek work on real ARM, we had to modify its managment of exceptions, its semaphore implementation, and add a cache activation routine.

3.1.1 Exception handling

One of the most meticulous thing to do when implementing an operating system is undoubtlessly the context save and restore while interrupted by an exception.

ARM architecture is well designed from this point of view. As we mentionned before, when an exception occurs, the CPU changes of operating mode from normal operating mode to exception mode. In the ARM implementation of Mutek, the system is always running in supervisor mode. The main advantage of this mode is the possibility of executing every instructions (privileged or not). For the sake of simplicity, we assume that only interruptions can occur (IRQ). Figure 13 shows the context before IRQ happens. Note that context register value will be marked with a "*" in the following steps.

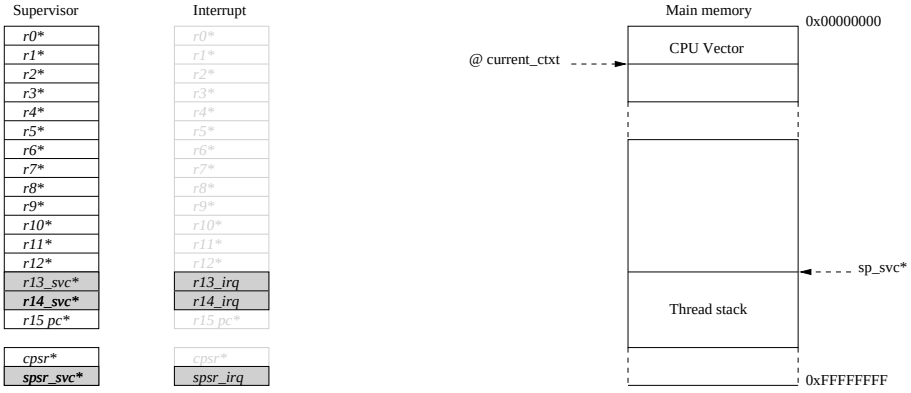


Figure 13: Before IRQ

When an interruption occurs, the CPU switches from supervisor mode to IRQ mode. The CPU runs then in IRQ mode, and IRQ and FIQ are disabled. To avoid the PC* register value loss, this register is saved in a banked register named LR_irq. For the same reason CPSR is saved in SPSR_irq (Saved Processor Status Register). LR_irq and SPSR_irq are banked register (*c.f.* section 2.1.1 Register file).

As you can realize from figure 14, registers r0* to r12* are not changed and are accessible in IRQ mode, we can save them as they are. The register r13_svc* and r14_svc* also called SP_svc* and LR_svc* are banked, they are not accessible any more. Finally the CPSR* is saved in SPSR_irq.

Before passing the control to the C function responsible of the IRQ managment, we must take care of storing all the registers, more precisely their value before IRQ occurs (*i.e.* all value marked with a "*"). To allow an easier context restore, we will store the context of a thread in its own stack. The trouble is that we have no access to the current top position of the stack (SP_svc*). The solution is to switch back into supervisor mode, but we have to save the two register value which would be lost otherwise. Figure 15 give an overview of the PC* and CPSR* storing process. You can also find the assembly code of this step.

SP_irq (r13_irq) could have been used as a pointer on the IRQ stack, but we will not use it like this. We use it like an empty general purpose register. This register allows us to store the PC* and CPSR* value at a known position in main memory, just behind the CPU vector.

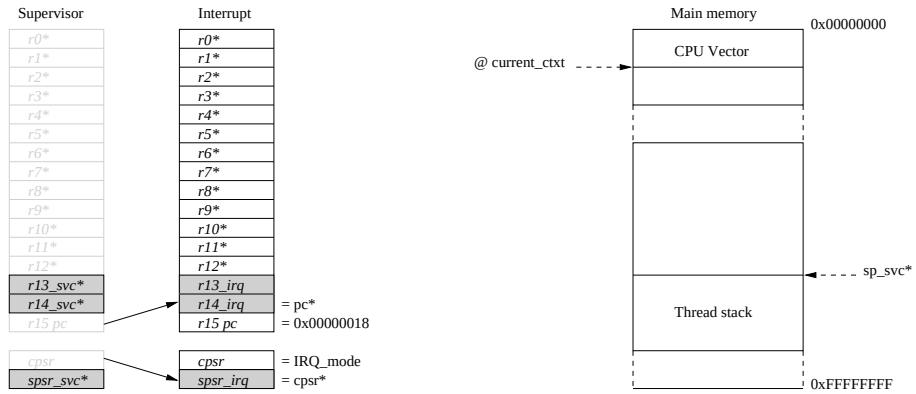
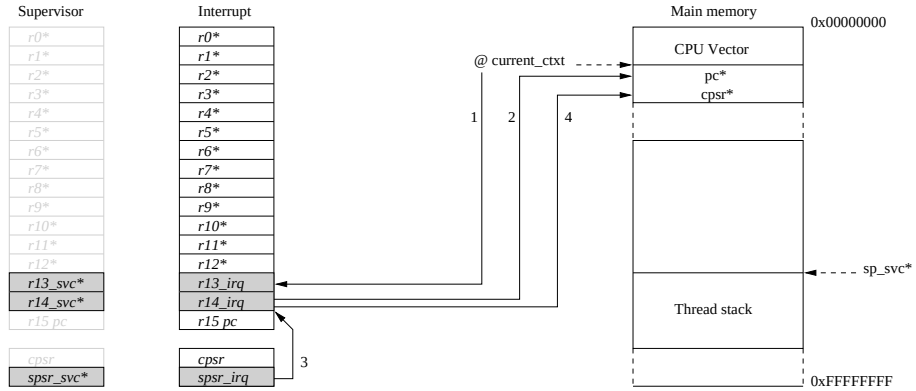


Figure 14: At IRQ raising




```
IRQ_Handler:
        @
        @ Jump here from exception ... sp and lr banked. :-(
        @

        ldr     r13, =current_ctxt          @ load temporary save address
        sub     lr, lr, #4
        str     lr, [r13], #4               @ Save last PC
        mrs     r14, spsr
        str     r14, [r13], #4              @ Save last CPSR
```

Figure 15: First phase of context storing : the saved registers PC* and CPSR*

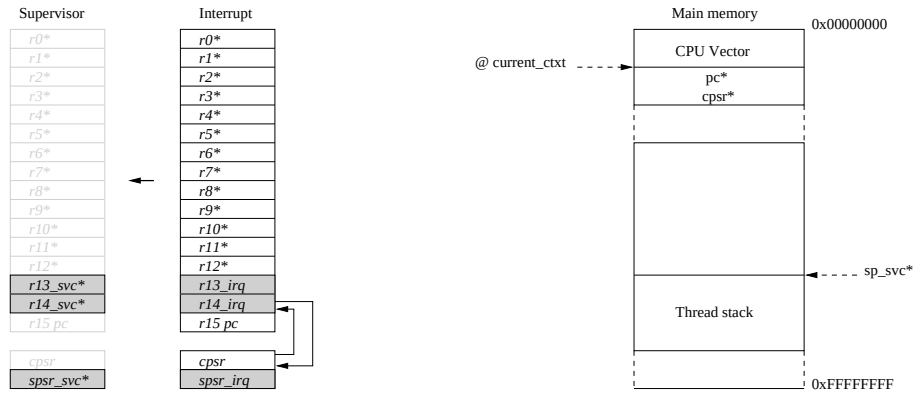
Once the two saved registers are stored in memory, we can switch back to supervisor mode, to retrieve the final memory address of context save. The procedure to get back to supervisor mode is simple, it is a read-modify-write sequence (figure 16). The modification is only an execution mode modification, then the CPU will continue to run with IRQ and FIQ disabled.

Now we can store register values in the stack of the process (thread). To do so we only need to allocate the context size in the stack (by removing this size to the value of the stack pointer), and write the thirteen first registers at their right place (see figure 17).

Last registers to be saved are SP_svc* (it is not yet available, but it is SP_svc plus context size), LR_svc* (it is still in place), PC* and CPSR* (these ones are stored in memory, but not at the right address). Figure 18 gives a description of the actions needed to bring these values in the registers r1 to r4.

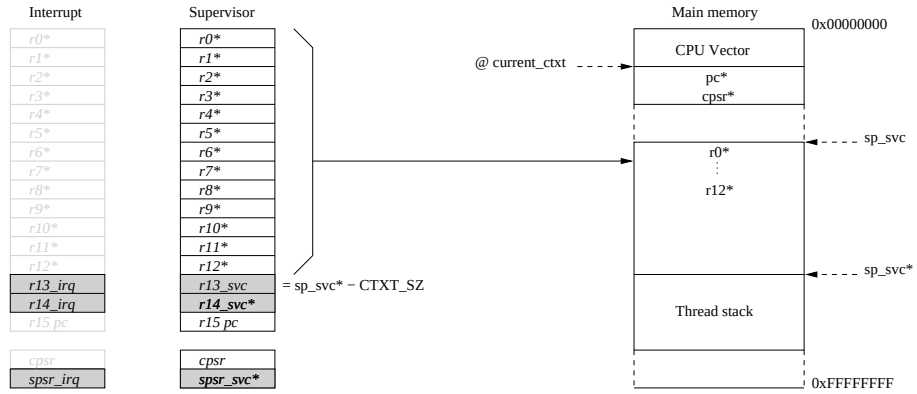
Once SP_svc* to CPSR* are placed in registers r1 to r4, we store them in memory (figure 19).

Now we have saved all registers. SP_svc points now to the top of stack, just before our saved con-




```
@-------------------------------------------------
@ Switch to SVC mode IRQ disabled
@-------------------------------------------------
mrs     r14, cpsr
bic     r14, r14, #PSR_MODE_MASK   @ clear execution mode
orr     r14, r14, #PSR_SVC_MODE    @ set supervisor mode
msr     cpsr, r14
```

Figure 16: Switch back to supervisor mode

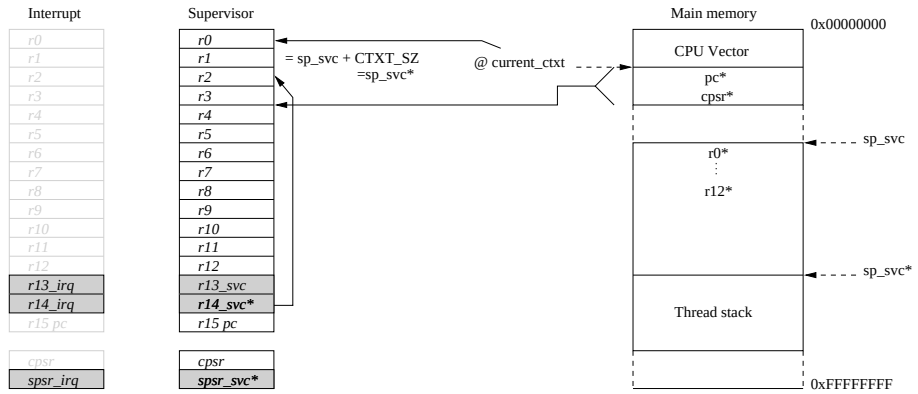



```
@-------------------------------------------------
@ Push context in thread stack
@-------------------------------------------------
sub     sp, sp, #CTXT_SIZE          @ CTXT size allocation
stmia   sp, {r0-r12}                @ save r0 though r12
```

Figure 17: First registers storing : r0* to r12*

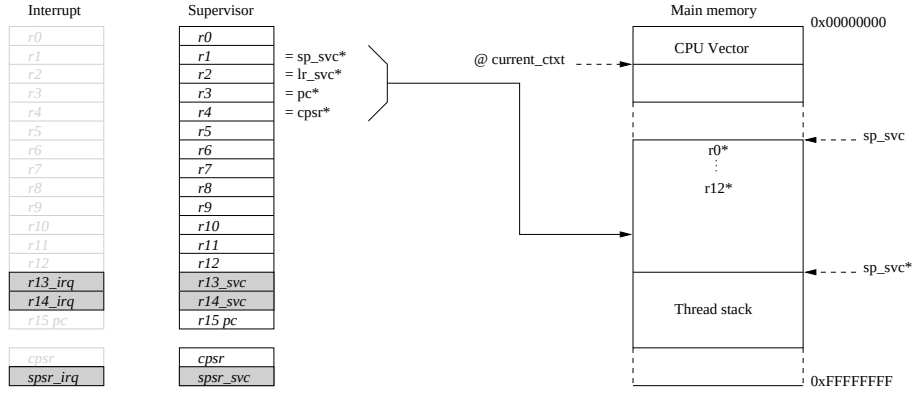
text. We can safely pass control to the C function in charge of identifying the source of interruption and launching the right interruption handler. This part of software is platform dependant, so it will be detailled further in this report. The situation before passing control to C function called `SystemInterrupt()` is depicted by figure 20.

Last step in the exception handling is to restore a context before getting back to the process interrupted. Note that this process is not necessarily the one interrupted just before, more precisely, this is not the case when the interruption source is the timer. In the last case it is likely to be a preemption tick. In the restore




```
ldr    r0, =current_ctxt          @ load temporary save address
add    r1, sp, #CTXT_SIZE         @ SP before IRQ
mov    r2, lr                     @ just for stm
ldmia  r0, {r3, r4}               @ grab back pc and cpsr.
```

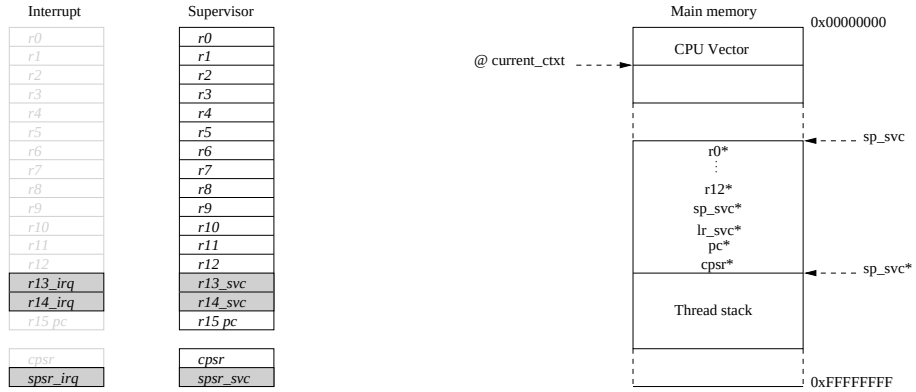
Figure 18: Last register retrieving : from phase one




```
add    r5, sp, #SP_OFF
stmia  r5, {r1-r4}
```

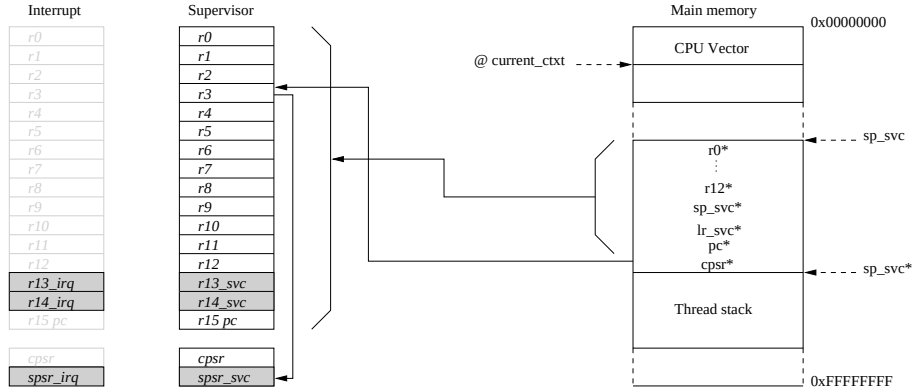
Figure 19: Last registers storing : SP*, LR*, PC* and CPSR*

process, we must underline the fact that we first need to read the CPSR value and put it in the SPSR register. The restore action and control passing is made in one instruction, a `ldmia`. Since we load the CP value and as we put a "^", all values are moved to the registers and the SPSR value is moved to the CPSR register. This means that the execution mode is updated (IRQ and FIQ are re-enabled). The reason why we put the modifications in SPSR and not directly in CPSR is that we would have enabled IRQ and FIQ exceptions in the current mode by modifying directly CPSR. By doing direct CPSR modifications, the context restore process could be interrupted and stopped in an unpredictable state. The instruction described before allow to make all actions without being interrupted. The full action sequence is shown on figure 21.




```
@ Jump and link to IT Handler
bl     SystemInterrupt
```

Figure 20: Context ready




```
ldr    r3, [sp, #PSR_OFF]
msr    spsr_cxsf, r3              @ put cpsr in SPSR
ldmia  sp, {r0-r15}^              @ restore Context
```

Figure 21: Restore process

3.1.2 Exception CPU Vector Relocation

We have a full exception handling routine which saves the context before passing control to higher level interruption handler. The CPU vector is the location where the CPU jumps when an exception occurs. As we mentionned before, each type of exception has its own offset. The vector looks like this :

```
excep:
       b       reset                @ 0x00 reset
       b       Undefined_Handler    @ 0x04
       b       SWI_Handler          @ 0x08
       b       Prefetch_Handler     @ 0x0C
       b       Abort_Handler        @ 0x10
       nop                          @ 0x14 not assigned
       b       IRQ_Handler          @ 0x18
       b       FIQ_Handler          @ 0x1c
       subs    pc, lr, #4
```

The CPU vector must be placed in memory at address `0x00000000`. In a simulation environment like Skyeye, there are no problem to load in memory an executable file at address `0x00000000` but on a real platform like CM 922T-XA10, the software used to load programs in memory does not allow this kind of action. Our solution to tackle this trouble is to use a code relocation routine which copies the CPU vector from an alternate position in memory to the `0x00000000` address region.

The relocator code is simple and integrated in our programs for now. Here is the algorithm :

```
except_reloc:
       stmfd  sp!,{r0-r12,lr}
       adr    r0, _b_table
       adr    r2, _e_table
       mov    r1, #0              @ excep_vect location : 0x00000000
1:     cmp    r0, r2
       ldrlt  r3, [r0], #4
       strlt  r3, [r1], #4
       blt    1b
2:     ldmfd  sp!,{r0-r12,lr}
       mov    pc, lr
```

`_b_table` and `_e_table` represents the addresses of the beginning and the end of the CPU vector. It should also include the low level handler described before.

Full source is here :

```
IRQ_Handler:
      @
      @ Jump here from exception ... sp and lr banked. :-(
      @

      ldr    r13, =current_ctxt
      sub    lr, lr, #4
      str    lr, [r13], #4              @Save last PC
      mrs    r14, spsr
      str    r14, [r13], #4             @Save last CPSR

      @-----------------------------------------------
      @ Switch to SVC mode IRQ disabled
      @-----------------------------------------------
      mrs    r14, cpsr
      bic    r14, r14, #PSR_MODE_MASK
      orr    r14, r14, #PSR_SVC_MODE
      msr    cpsr, r14

      @-----------------------------------------------
      @ Push context in thread stack
      @-----------------------------------------------
      sub    sp, sp, #CTXT_SIZE         @ thus we can store sp immediately
      stmia  sp, {r0-r12}               @ save r0 though r12
      ldr    r0, =current_ctxt
```

```
        add    r1, sp, #CTXT_SIZE         @ SP before IRQ
        mov    r2, lr                     @ just for stm
        ldmia  r0, {r3, r4}               @ grab back pc and cpsr.
        str    sp, [r0]
        add    r5, sp, #SP_OFF
        stmia  r5, {r1-r4}

        @ Jump and link to IT Handler
        bl     SystemInterrupt

        ldr    r3, [sp, #PSR_OFF]
        msr    spsr_cxsf, r3              @ put cpsr in SPSR
        ldmia  sp, {r0-r15}^              @ restore Context
```

3.1.3 Spinlocks Implementation

The implementation of the semaphores is based on spinlocks. Unfortunately these spinlocks were not working due to little implementation mistakes. In short, the previous implementation was based on multiple `asm()` calls, to include the assembly langage of the spinlock. On top of that some relative jumps was made in these instructions. Unfortunately `asm()` adds some instructions before and after the assembler instructions given in arguments. Thus the asm sequence was not exactly the same than the one written in the C file, some instructions were inserted and relatives branches did not jump to the right place.

We took the decision to reimplement the spinlocks. This reimplementation is really simple. Like the previous, this one is based on the atomic instruction `swp` (swap), which allows us to read the value of the spin-lock and replace it by another value in one non interruptible instruction. To implement spinlock algorithm, the value written is 1 and the value read must be 0 to continue. If it is not the case, we repeat the swap until it is a 0. To avoid relative branches error to occur in the future, we used labeled jumps, then the assembler is in charge of putting the right address in the branch instruction. And finally, we put all assembly instruction in one unique `asm()` call.

To release the spinlock we only need to write a 0 at the spinlock memory address.

Here is the source code :

```
#define SEM_LOCK(semaddr)                                          \
     do {                                                          \
       asm(                                                        \
        "   stmdb sp!, {r4-r6}   @                        \n"   \
        "   mov   r6, %0         @load the semaddr in a reg\n"   \
        "   mvn   r5, #1         @load 1                  \n"   \
        "1: swp   r4, r5, [r6]   @                        \n"   \
        "   cmp   r4, #0         @                        \n"   \
        "   bne   1b             @loop                    \n"   \
        "   ldmia sp!, {r4-r6}   @                        \n"   \
        ::"r"(semaddr):"r4","r5","r6");                           \
     } while(0)

#  define SEM_UNLOCK(semaddr)                                     \
     do {                                                          \
       asm(                                                        \
        "   mov r6, %0           @                         \n"  \
        "   mov r5,#0            @                         \n"  \
        "   str r5, [r6]         @                         \n"  \
        ::"r"(semaddr):"r4","r5","r6");                           \
     } while(0)
```

3.1.4 Caches Activation

Instruction and Data caches can be activated thanks to a control register in the configuration coprocessor `CP15` register `c1`. For instruction cache, no particular attention is needed before activation. But as far as the data cache is concerned, we have to be careful because some part of the address space must not be cached and buffered (when written). For example, these parts correspond to the memory mapped physical registers. We can tell the CPU where (on which part of the address space) to use or not data cache and write buffer. This information is placed in the page (or section) table entries. The only mean to use the data cache is thus to activate the MMU.

In our case, we do not need a complex memory mapping, as the one used in the Linux kernel. We decided to use an identity mapping (it is used in the early deflation stage of the Linux kernel). In that aim, we choose to build a page directory filled with only section descriptions (*c.f.* section 2.1.3). We only put cache-able and buffer-able informations on the 128 first section descriptors since we have an amount of 128 Mo of main memory.

```
__setup_mmu:
                bic     r3, r3, #0xff           @ Align the pointer
                bic     r3, r3, #0x3f00
/*
 * Initialise the page tables, turning on the cacheable and bufferable
 * bits for the RAM area only.
 */
                mov     r0, r3
                mov     r8, #0x0                @ start of RAM
                add     r9, r8, #0x08000000     @ a reasonable RAM size
                mov     r1, #0x12
                orr     r1, r1, #3 << 10
                add     r2, r3, #16384
1:              cmp     r1, r8                  @ if virt > start of RAM
                orrhs   r1, r1, #0x0c           @ set cacheable, bufferable
                cmp     r1, r9                  @ if virt > end of RAM
                bichs   r1, r1, #0x0c           @ clear cacheable, bufferable
                str     r1, [r0], #4            @ 1:1 mapping
                add     r1, r1, #1048576
                teq     r0, r2
                bne     1b
```

Once the page directory is built, last actions are to activate the MMU, the instruction and data caches. To avoid bad surprises, we invalidate the TLBs and caches, since they could contain expired translations and cache lines.

```
__cache_on:
                mov     r12, lr
                bl      __setup_mmu             @ build page tables
                mov     r0, #0
                mcr     p15, 0, r0, c7, c10, 4  @ drain write buffer
                mcr     p15, 0, r0, c8, c7, 0   @ flush I,D TLBs
                mcr     p15, 0, r0, c7, c7, 0   @ invalidate caches
                mrc     p15, 0, r0, c1, c0, 0   @ get control reg
                orr     r0, r0, #0x5000         @ enable I-cache enable,
                                                @ RR cache replacement
                orr     r0, r0, #0x0005         @ enable D-Cache and MMU
                mvn     r1, #0
                mcr     p15, 0, r3, c2, c0, 0   @ load page table pointer
                mcr     p15, 0, r1, c3, c0, 0   @ load domain access control
```

```
mcr     p15, 0, r0, c1, c0, 0   @ load control register
mov     r0, #0
mcr     p15, 0, r0, c8, c7, 0   @ flush I,D TLBs
mov     pc, r12
```

3.2 Specifi c Device Drivers

3.2.1 Serial device driver

First device to require a driver implementation is the serial interface for character terminal printings. Once this driver is developped, we will be able to know if Mutek is effectively working.

Driver API: This driver proposes to the remainder of Mutek the following functions (`console.h`) :

```
void init();
inline void writec(const char c);
inline void writes(const char* s);
inline char readc();
```

The function names help to understand their aim. We will not give more details about these.

ARM Integrator CM922T-XA10: In the platform specific part of the driver we must implement functions introduced above :

```
void uart00_init();
inline void uart00_writec(const char c);
inline void uart00_writes(const char* s);
inline char uart00_readc();
```

The function names are quite expressive, but we will give some details. `uart00` is the name given to the UART device driver in the Linux kernel. We keep the same name, but there are no particular reason.

On top of defining these functions, the driver also needs to define the register locations and usage. A structure describe the registers mapping like this :

```
struct cm922txa10_uart00
 {
  /* I/O register
   * */
  unsigned long rsr;   /* UART00 Receive Status Register */
  unsigned long rds;   /* UART00 Received Data Status */
  unsigned long rd;    /* UART00 Received Data */
  unsigned long tsr;   /* UART00 Transmit Status Register */
  unsigned long td;    /* UART00 Transmit Data */
  unsigned long fcr;   /* UART00 FIFO Control Register */
  unsigned long ier;   /* UART00 Interrudpt-Enable Set and Clear */
  unsigned long iec;
  unsigned long isr;   /* UART00 Interrupt Status Register */
  unsigned long iid;   /* UART00 Interrupt ID */
  unsigned long mc;    /* UART00 Mode-Configuration */
  unsigned long mcr;
  unsigned long msr;
  unsigned long div_lo; /* UART00 Divisor */
  unsigned long div_hi;
};
```

The implementation of the function described before is highly dependent on this structure. For exemple, here is the implementation of the function `writec()` :

```
#define TX_READY() ((CM922TXA10_UART00_BASEP->tsr & 0x1F) < 16)

inline void uart00_putc(const char c)
{
  //unsigned int status;
  /* Wait until there is space in the FIFO */
  while(!TX_READY());
  /* Send the character */
  CM922TXA10_UART00_BASEP->td = (unsigned int)c;
}

inline void uart00_writec(const char c)
{
  if (c == '\n')
    uart00_putc('\r');

  uart00_putc(c);
}
```

The general behavior of the UART is that we must wait for empty space in the fifo by checking a status register, and then write the character in the fifo by the mean of a TX register.

ATMEL AT91RM9200: As for CM922T-XA10, we define the following functions :

```
void usart_init();
inline void usart_writec(const char c);
inline void usart_writes(const char* s);
inline char usart_readc();
```

The register mapping is the following in AT91RM9200 USART :

```
struct at91rm92_usart
{
  /* I/O register
   * */
  unsigned long cr; /* control */
  unsigned long mr; /* mode */
  unsigned long ier; /* interrupt enable */
  unsigned long idr; /* interrupt disable */
  unsigned long imr; /* interrupt mask */
  unsigned long csr; /* channel status */
  unsigned long rhr; /* receive holding */
  unsigned long thr; /* tramsmit holding  */
  unsigned long brgr; /* baud rate generator */
  unsigned long rtor; /* rx time-out */
  unsigned long ttgr; /* tx time-guard */
  unsigned long fidi;
  unsigned long ner;
  unsigned long us_if;

  unsigned long sysflg;
};
```

With this informations, the `writec()` function implemtentation is the following :

```
inline void usart_wait()
{
  while (AT91RM92_USART_BASEP->csr != (AT91RM92_CSR_TXEMPTY | AT91RM92_CSR_TXRDY))
    {
```

```
      /* busy wait */
    }
}

inline void usart_writec(const char c)
{
  AT91RM92_USART_BASEP->thr = c;
  usart_wait();
}
```

The philosophy here is to wait until the fifo is empty after having put a character in it. Then when we want to write something in it we know that it is allways empty. This implementation is different from the previous one, but it could have be the same since UART and USART have close mechanisms.

3.2.2 Interrupt Controller Device Driver

The second step in device driver implementation is the Interruption Controller (ITC) since this is a first step in the direction of preemption implementation.

Driver API: The generic API for this driver is made of the following functions :

```
void init      ();
void mask_irq  (unsigned int irq);
void unmask_irq(unsigned int irq);
```

The meaning of these function is quite simple. When we want to enable an irq, we must tell the ITC to set its mask. This is the aim of `mask_irq()`. `unmask_irq()` has the opposite aim, when an IRQ occurs we must mask it to continue to work without being interrupted every cycle

ARM Integrator CM922T-XA10: The driver implementation of the EPXA10 interruption controller `itc00` contains exactly the function introduced before :

```
void cm922txa10_itc00_init(void);
void cm922txa10_itc00_mask_irq(unsigned int irq);
void cm922txa10_itc00_unmask_irq(unsigned int irq);
```

The device regiter description is also made with a structure looking like this :

```
struct cm922txa10_itc00
{
  unsigned long ims;         /* Mask Set */
  unsigned long imc;         /* Mask Clear */
  unsigned long iss;         /* Source Status */
  unsigned long irs;         /* Request Status */
  unsigned long iid;         /* Interrupt ID */
  unsigned long ppr;         /* PLD priority */
  unsigned long mod;         /* PLD Mode */
  unsigned long unused[25]; /* unused */
  unsigned long prio[17];    /* Priorities */
};
```

For exemple, the implementation of the function `maks_irq()` for `itc00` is really simple :

```
void cm922txa10_itc00_mask_irq(unsigned int irq)
{
  CM922TXA10_ITC00_BASEP->ims = (1 << irq);
}
```

We only need to set a register value (`ims` interrupt mask set).

The functions described till there are low level interruption controller functions. An extra function need to be completed, which is the interruption handling function. This function will be called by the exception routine, when an interruption will occur. The name of this handler is fixed to `SystemInterrupt()` and its definition must be placed in the `it.c` file in the platform directory. Briefly, this function reads the status register of the interruption controller, and calls the specific interruption handler, which must be registered by the driver of the source device.

ATMEL AT91RM9200: The implementation of the ITC in AT91RM9200 as a little specificity. On top of the generic functions described earlier, we need two extra functions `set_irq()` and `clear_irq()` :

```
void at91rm92_aic_init      ();
void at91rm92_aic_mask_irq  (unsigned int irq);
void at91rm92_aic_unmask_irq(unsigned int irq);
void at91rm92_aic_set_irq   (unsigned int irq);
void at91rm92_aic_clear_irq (unsigned int irq);
```

The register mapping is the following :

```
struct at91rm92_aic
{
  unsigned long aic_smr[32];     /* 0x000 Source Mode   register             (RW) */
  unsigned long aic_svr[32];     /* 0x080 Source Vector register             (RW) */
  unsigned long aic_ivr;         /* 0x100 Interrupt Vector register          (RO) */
  unsigned long aic_fvr;         /* 0x104 Fast Interrupt Vector register     (RO) */
  unsigned long aic_isr;         /* 0x108 Interrupt status register          (R0) */
  unsigned long aic_ipr;         /* 0x10C Interrupt pending register         (RO) */
  unsigned long aic_imr;         /* 0x110 Interrupt Mask register            (RO) */
  unsigned long aic_cisr;        /* 0x114 Core interrupt status register     (RO) */
  unsigned long reserved[2];     /* 0x118 & 0x11C                                 */
  unsigned long aic_iecr;        /* 0x120 Interrupt Enable Command register  (WO) */
  unsigned long aic_idcr;        /* 0x124 Interrupt Disable Command register (WO) */
  unsigned long aic_iccr;        /* 0x128 Interrupt Clear Command register   (WO) */
  unsigned long aic_iscr;        /* 0x12C Interrupt Set Command register     (WO) */
  unsigned long aic_eoicr;       /* 0x130 End of Interrupt Command register  (WO) */
  unsigned long aic_spu;         /* 0x134 Spurious Interrupt Vector register (RW) */
  unsigned long aic_dcr;         /* 0x138 Debug Control register             (RW) */
  unsigned long reserved1;
};
```

As for CM922T-XA10 the `mask_irq()` operation is only a register value write :

```
void at91rm92_aic_mask_irq(unsigned int irq)
{
  unsigned long mask = 1 << (irq);
  AT91RM92_AIC_BASEP->aic_idcr = mask;
}
```

The definition of `SystemInterrupt()` must be adapted to the specific registers of this interruption controller. It is placed in the folder of the platform.

3.2.3 Timer device driver

Final step to get preemption working is to develop a driver for the timer. The main task of this driver is to configure the timer to interrupt the CPU at a predefined interval of time.

Driver API: The API of the timer driver is quite complexe in Mutek since it must manage with multiple processor synchronization. The full list of primitives is the following :

```
void timerSetInterruptEnable();
void timerSetInterruptDisable();
void timerSetPeriod(unsigned int v);
void timerSetSynchronizedPeriod(unsigned int v);
int  timerGetPeriod();
int  timerGetState();
void timerSetCountingRun();
void timerResetInterrupt(int procid);
void timerSetSynchronizedMode();
void timerInterruptInit(unsigned int);
void timerInterruptHandler(void);
```

Briefly, these functions allows to enable and disable the interruption assertion in the hardware timer, to configure the period between two ticks and to handle the interruption asserted. Some of these functions have only a sense when we work on multi-processor platforms.

ARM Integrator CM922T-XA10: In a first implementation, we do not forsee the integration of multiple processor. This choice allows us to declare most of the function presented above as empty functions.

The currently implemented functions are :

```
void timerSetInterruptEnable();
void timerSetInterruptDisable();
void timerSetPeriod(unsigned int v);
void timerResetInterrupt(int procid);
```

The `timer00`, which is the name of the timer available on the EPXA10, we need to set a control bit to tell it to start (or stop). We decided to implement two extra functions whose aims are to start and stop the timer.

```
void timerStart(void);
void timerStop(void);
```

The registers of the timer are mapped in a structure, as for other peripherals. The definition of this structure looks like the following one:

```
struct cm922txa10_timer00 {

  unsigned long tcr;  /* Timer Control Register */
  unsigned long unused_1[3];
  unsigned long tpre; /* Timer Prescale */
  unsigned long unused_2[3];
  unsigned long tl;   /* Timer Limit */
  unsigned long unused_3[3];
  unsigned long tr;   /* Timer Read */

};
```

With all these definition, the timer is ready for usage. But one important thing still needs to be implemented to make preemption work. This last step is the interruption handler for timer interruptions. This handler will only disable the interruption and call the preemption facilities. This part of the code is left as it is since it works, once semaphores (more precisely spinlocks used in the semaphore implementation) are fixed.

ATMEL AT91RM9200: The functions implemented for the AT91 are the same than the one implemented for the EPXA10. Only the register mapping is different.

```
struct at91rm92_st {
  unsigned long      st_cr;        // Control Register
  unsigned long      st_pimr;      // Period Interval Mode Register
  unsigned long      st_wmr;       // Watchdog Mode Register
  unsigned long      st_rtmr;      // Real-time Mode Register
  unsigned long      st_sr;        // Status Register
  unsigned long      st_ier;       // Interrupt Enable Register
  unsigned long      st_idr;       // Interrupt Disable Register
  unsigned long      st_imr;       // Interrupt Mask Register
  unsigned long      st_rtar;      // Real-time Alarm Register
  unsigned long      st_crtr;      // Current Real-time Register
};
```

This timer do not need start and stop functions. The remainder of the procedure is identical to the one presented for CM922T-XA10.

4 Mutek Design and Programming Model

4.1 CPU Vector Relocalisation or Mapping

In the Mutek implementation section, we presented the current implementation of the CPU vector. In fact, we build a CPU vector in assembler language and copy it at execution time to the right place (`0x00000000`). This solution is the only possible solution when we do not use the MMU, as we have no mapping possibilities. Indeed, with the use of MMU, other solutions are available. Among them, we can use the CPU vector remapping option, that is to say that the CPU vector can be place at address `0xFFFF0000`, which is impossible whithout MMU.

In that case an interresting option is available. We can map the memory region of the CPU vector at address `0xFFFF0000` and add a Null pointer trap at address `0x00000000` avoiding to reset the platform each time we make a null pointer access. This is made by declaring an invalid page at location `0x00000000`. Each instruction prefetch or data access at this address will generate an abort exception (prefetch abort or data abort).

The two solutions can be ported into Mutek, since we use the MMU in order to use the data cache.

4.2 Mutek/ARM execution mode

As we mentionned before Mutek is running in supervisor mode. This mode must have been chosen because all instructions are executable since it is a privileged mode of execution. In the description of context saving, we saw that two registers are not accessible (SP and LR) while handling an IRQ in IRQ mode. This is not fully true. In fact they are not accessible because we run in supervisor mode and SP and LR are banked (SP_svc and LR_svc).

In fact the ARM instruction set allow retrieving normal SP and LR with a special bit in the instructions `ldm` and `stm`. To keep the advantage of running in privileged mode to avoid mode switching when we want execute privileged instructions we should use the system mode. This mode of execution, present on ARM v4 architecture, is a privileged mode of execution which uses no banked registers.

This could have great advantages for us, because the exception handler can be simplified, and all context storing can be made in IRQ mode.

Another great advantage is noticeable. If we envision the use of software interrupts, or some reset handling, the current configuration will not allow them. The CPU automatically update the LR_svc with the PC value, and as the exception mode of software interrupts or reset is supervisor, the saved PC will erase the process LR. LR value will then be lost.

We beleive that these two reasons should promote the use of *system mode* to run the Mutek operating system instead of the *supervisor mode*.

4.3 Mutek Programming Model

The Mutek operating system is a very lightweight implementation of the POSIX thread library. As such, the Mutek programming model is based on a very thin hardware abstraction layer so that threads can have access to the complete address space. Mutek can already support a wide range of applications. The next turning point we see on the Mutek development cycle is to know whether the device drivers and high level support within the kernel code or should be part of the application code. The timer and interrupt controller is the only hardware dependent part of the kernel, all other drivers can be written in POSIX threads that will be scheduled with other application threads. This decision has a big impact on the I/O functions provided by the Mutek `libc`. The `libc` should be re-written to refine the Mutek programming model.

References

[1] Soclib simulation environment. Available online, `http://soclib.lip6.fr/`, 2005.

[2] ARM. Arm integrator/cm922t-xa10 user guide. Available online, http://www.arm.com/pdfs/DUI0184A_CM922T.pdf, October 2005.

[3] Atmel. Atmel at91rm9200 platform. Available online, http://www.atmel.com/dyn/products/product_card.asp?part_id=2983, October 2005.

[4] I. Augé, F. Donnet, P. Gomez, D. Hommais, and F. Pétrot. Disydent: a pragmatic approach to the design of embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, Paris, France, March 2002.

[5] Nicolas Fournel, Antoine Fraboulet, and Paul Feautrier. Booting and Porting Linux and uClinux on a new platform. Research Report 2006-08, LIP, ENS-Lyon, February 2006. 28 pages.

[6] Mutek Operating System. Disydent web site. Available online, http://www-asim.lip6.fr/recherche/disydent/, October 2005.

[7] Andrew N.Sloss, Dominic Symes, and Chris Wright. *ARM System Developer's Guide*. Morgan Kaufmann, 2004. ISBN 1-55860-874-5.

[8] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition, 2001. ISBN 0-201-73719-1.

[9] Skyeye. Skyeye web site. Available online, http://skyeye.sourceforge.net/, October 2005.

A Core Toolchain compilation Script

The DISYDENT framework comes with gcc 3.2, during our experiments we needed to upgrade to gcc-3.3. The configuration script to build gcc need to be updated to the new configuration gcc scheme. This appendix presets the basic steps used to build a cross compiler for Mutek/ARM.

```
CXTOOLS=/usr/local/cross_toolchain/arm-core/gcc-3.3.6-none/arm-9tdmi-linux-gnu
BUILD_DIR=/home/nfournel/tmp/gcc_crossbuild
TARGET=arm-9tdmi-linux-gnu

GCC_VER=3.3.6
BINUTILS_VER=2.16.1

#binutils
#========

cd $BUILD_DIR/src

if test ! -d binutils-$BINUTILS_VER; then
    tar jxvf binutils-$BINUTILS_VER.tar.bz2
fi
mkdir -p $BUILD_DIR/build/binutils-$BINUTILS_VER
cd $BUILD_DIR/build/binutils-$BINUTILS_VER
$BUILD_DIR/src/binutils-$BINUTILS_VER/configure   \
    --prefix=$CXTOOLS --target=$TARGET
make ; make install


#gcc
#===

export PATH=$CXTOOLS/bin:$PATH

cd $BUILD_DIR/src
if test ! -d gcc-$GCC_VER ; then
    tar jxvf gcc-$GCC_VER.tar.bz2
fi
mkdir -p $BUILD_DIR/build/gcc-$GCC_VER
cd $BUILD_DIR/build/gcc-$GCC_VER

$BUILD_DIR/src/gcc-$GCC_VER/configure --prefix=$CXTOOLS          \
       --srcdir=$BUILD_DIR/src/gcc-$GCC_VER                      \
       --target=$TARGET                                        \
       --enable-languages=c,c++ --with-gnu-as --with-gnu-ld    \
       --disable-shared --disable-multilib --disable-threads   \
       --disable-libgcj --disable-nls --without-newlib         \
       --disable-libstdcxx-v3 --with-cpu=arm9tdmi ;

export ALL_TARGET_MODULES=""
export CONFIGURE_TARGET_MODULES=""
export INSTALL_TARGET_MODULES=""
make -e; make -e install
```

B AT91RM9200 ldscript

```
/*
 * Mutek linker script for ARM AT91RM9200
 * Antoine Fraboulet
 *
 */

MEMORY
{
   init            : ORIGIN = 0xc0000000, LENGTH = 0x00010000 /* 65 KB   */
   text            : ORIGIN = 0xc0010000, LENGTH = 0x00080000 /* 512 KB  */
   data            : ORIGIN = 0xc0090000, LENGTH = 0x00100000 /* 1 MB    */
   /* 0xc0190000 - 0xc01a0000 is used for stack () */
   reset           : ORIGIN = 0xc01a0000, LENGTH = 0x00010000
   excep           : ORIGIN = 0xc01b0000, LENGTH = 0x00010000
}

EXTERN(reset)
EXTERN(excep)
ENTRY(excep)

SECTIONS
{

   .init  : {
      /home/antoine/projets/armgcc/mutek/mutek-oes/lib/libhandler.a(.init)
   } > init
   .text : {
      . = ALIGN(0x4) ;
      *(.text)
      *(.gnu.linkonce.t*)
      *(.glue_7t)
      *(.glue_7)
      . = ALIGN(0x10) ;
      _etext = . ;
   } > text
   .excep : {
      /home/antoine/projets/armgcc/mutek/mutek-oes/lib/libhandler.a(.excep)
   } > excep
   .reset : {
      /home/antoine/projets/armgcc/mutek/mutek-oes/lib/libhandler.a(.reset)
   } > reset

   .data  : {
      *(.rodata) *(.rodata.str1.4)
      . = ALIGN(4);
      *(.data) *(.lit8) *(.lit4) *(.sdata)
      __sem_addr = .;
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
```

```
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      _edata = . ;
      *(.sbss) *(.scommon) *(.bss) *(COMMON)
      _end = . ;
   } > data

   /* Semaphore address */
   /* __sem_addr         = 0x01110100; */

   /* Required, but useful only when dealing with scratch pad memories */
   __spm_addr = 0;
   __spm      = 0;

   /* Default stack address, for ALL processors */
   /* the stack starts above the data section   */
   __irq_stack_addr   = 0xc0190000       + 0x400;
   __fiq_stack_addr   = __irq_stack_addr + 0x400;
   __und_stack_addr   = __fiq_stack_addr + 0x400;
   __abt_stack_addr   = __und_stack_addr + 0x400;
   __svc_stack_addr   = __abt_stack_addr + 0x1000;

   /* peripheral adress */
   __tty_addr         = 0xfffc0000; /* USART0       */
   __itc_addr         = 0xfffff000; /* AIC          */
   __tmr_addr         = 0xfffffd00; /* System Timer */

   __processor_number = 1;
}

SEARCH_DIR (/home/antoine/projets/armgcc/mutek/mutek-oes/lib)
GROUP      (libc.a libhandler.a libpthread.a)
```

C Integrator CM922T-XA10 ldscript

```
/*
 * Mutek linker script for ARM Integrator CM922T-XA10
 *
 */

MEMORY
{
   init            : ORIGIN = 0x00020000, LENGTH = 0x00010000 /* 65 KB   */
   page_table      : ORIGIN = 0x00100000, LENGTH = 0x00004000 /* 16 KB   */
   text            : ORIGIN = 0x00200000, LENGTH = 0x00100000 /* 1 MB  */
   data            : ORIGIN = 0x00500000, LENGTH = 0x00100000 /* 1 MB    */
   /* 0xc0190000 - 0xc01a0000 is used for stack () */
   reset           : ORIGIN = 0x00600000, LENGTH = 0x00010000

   /* excep           : ORIGIN = 0x001b0000, LENGTH = 0x00010000  */
}

EXTERN(reset)
EXTERN(excep)
ENTRY(excep)

SECTIONS
{

   .init  : {
      /home/nfournel/CVS_eos/mutek-oes/bin-at91/lib/libhandler.a(.init)
   } > init

   .excep : {
      /home/nfournel/CVS_eos/mutek-oes/bin-at91/lib/libhandler.a(.excep)
   } > init

   .pgtable  : {
      __pgt_base = . ;
   } > page_table

   .text : {
      . = ALIGN(0x4) ;
      *(.text)
      *(.gnu.linkonce.t*)
      *(.glue_7t)
      *(.glue_7)
      . = ALIGN(0x10) ;
      _etext = . ;
   } > text

   .reset : {
      /home/nfournel/CVS_eos/mutek-oes/bin-at91/lib/libhandler.a(.reset)
   } > reset

   .data  : {
      *(.rodata) *(.rodata.str1.4)
```

```
      . = ALIGN(4);
      *(.data) *(.lit8) *(.lit4) *(.sdata)
      __sem_addr = .;
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      LONG(0)
      _edata = . ;
      *(.sbss) *(.scommon) *(.bss) *(COMMON)
      _end = . ;
   } > data

   /* Semaphore address */
   /* __sem_addr         = 0x01110100; */

   /* Required, but useful only when dealing with scratch pad memories */
   __spm_addr = 0;
   __spm      = 0;

   /* Default stack address, for ALL processors */
   /* the stack starts above the data section   */
   __irq_stack_addr   = 0x00390000       + 0x400;
   __fiq_stack_addr   = __irq_stack_addr + 0x400;
   __und_stack_addr   = __fiq_stack_addr + 0x400;
   __abt_stack_addr   = __und_stack_addr + 0x400;
   __svc_stack_addr   = __abt_stack_addr + 0x1000;

   /* peripheral adress */
   __tty_addr         = 0x0B000280; /* UART00       */
   __itc_addr         = 0x0B000C00; /* ITC00        */
   __tmr_addr         = 0x0B000200; /* Timer0       */

   __processor_number = 1;
}

SEARCH_DIR (/home/nfournel/CVS_eos/mutek-oes/bin-at91/lib)
GROUP      (libc.a libhandler.a libpthread.a)
```