# AVR32 port of the OKL4 microkernel

Adriana Drăghici, Marius Sandu-Popa, Andrei Voinescu
*Automatic Control and Computers Faculty*
*University Politehnica of Bucharest*
*Bucharest, Romania*
*Email: adriana.draghici@cti.pub.ro, sandupopamarius@gmail.com, voinescu.andrei@gmail.com*

***Abstract*—The present study addresses the differences in architecture between AVR32 and ARM from a micro-kernel's point of view. Different topics are discussed and an approach to handling the differences in architecture for each of these topics is proposed.**

***Keywords*-microkernel; avr32; port;**

## I. Introduction

The demand for more complex functionality has pushed embedded systems towards more powerful platforms, 32-bit (sometimes even 64-bit) general-purpose processors. Such devices have requirements (real-time properties, reliability, security) that are quite different from classical embedded systems, and can only be supported by a well designed operating system.

The L4 micro-kernel provides a minimal and efficient basis for constructing operating system software for a broad range of embedded devices. Originally implemented as a highly tuned Intel i386-specific assembly language code, the L4 micro-kernel has seen extensive development in a number of directions, both in achieving a higher grade of platform independence and also in improving security, isolation, and robustness.

One of the most important L4 implementation for embedded devices is the OKL4 micro-kernel developed by Open Kernel Labs company. It offers a range of features and capabilities: virtualization, small memory footprint, extensible and maintainable, real-time capability and low performance overhead, freely available source code.

The OKL4 currently targets ARM systems, supporting both v5 and v6 architectures and 926, 1136, 920, XScale platforms. We aim to port this micro-kernel to the AVR32 architecture. Similar to ARM, the AVR32 architecture is based on a RISC instruction-set, includes a Memory Management Unit (MMU), Java hardware acceleration and supports operating systems like Linux.

Porting the OKL4 micro-kernel to a new architecture requires the following implementation stages:

- implement the architecture-specific code in **arch/**
- implement the platform-specific code in **platform/**
- implement the necessary device drivers

For our test architecture we chose Atmel's NGW100 platform with an AVR32B microprocessor from the AP700x family.

In this paper we describe in detail the stages of porting the OKL4 micro-kernel, the differences between ARM and AVR32 architectures and the impact these differences have on the micro-kernel's performance.

## II. Differences in Instruction Set Architecture

The AVR32 Architecture bears great resemblance to the ARM architecture in general, being a 32-bit load/store architecture. The difference lies however in details, details that make the AVR32 architecture to be easy to use by software running on top of it.

The architecture is made of instructions of variable instruction length, leading to great decreases in code size. Most of the instructions take one cycle to execute as well, making the AVR32 architecture a good setting for code that is both fast and small.

### *A. General Purpose Registers*

Both instruction sets are completely orthogonal, with 15 general-purpose registers. Although each of these can be used in instructions involving registers, three of these have a special meaning:

- R15 is also PC, the Program Counter.

- R14 is LR, the Link Register (holds the address to which the code must return.
- R13 is SP, the Stack Pointer.

Additionally, R12 is considered to hold the return value of a subroutine on AVR32, while on ARM R0 is used for that purpose. To maintain coherency we mirrored the use of explicit register numbers (R12 → R1, R11 → R2, and so on).

## *B. Privileged and Unprivileged Modes*

Privileged modes are organized differently: On ARM, we have User and Supervisor mode, along with Abort, Exception and Undefined. For interrupts there are two possible modes, either normal interrupt or Fast Interrupt Mode (FIQ). FIQ has at its disposal several banked registers (R8-R13), so that an interrupt handler need not worry about saving these registers on stack.

On AVR32 additional modes are available. Interrupts can have one of four possible levels and priorities between these levels along with assignment can be configured through a set of system registers. The Architecture allows for banking of R0-R12 in each interrupt mode, but the processor family used in this study only has banked registers for INT3 interrupt level, registers R8 through R12. This effectively transforms INT3 into the equivalent of FIQ.

## *C. Register File*

Each operating mode on ARM has its own set of special registers, SP, LR and PC. On AVR32, transition between modes is considered more important than modes, such that there are two special registers in the privileged modes (any except Application/User Mode), RSR and RAR, that retain the Program Counter and the Status register of the previous mode.

As such, transition between modes is easier on the AVR32 due to the well-defined mechanisms for entry and exit made available by the architecture. Transition to Supervisor mode, for example, can only be made through the 'scall' instruction, which saves the current PC in RAR_SUP and SR in RSR_SUP and modifies the SR to mirror the change in operating mode.

Also a notable difference is the existence of a common register stack pointer for all privileged modes, which permits code in Exception Mode for example to manipulate the system stack transparently, without need for temporary switches between modes.

Below is an example of how this switch to supervisor mode is no longer necessary in AVR32 in the case of IRQ exception. In addition,

```
        srsdb   r13_svc!
```

(save return stack with decrement before) is no longer needed because the status register and link register from application/user mode are saved automatically in RAR_EX and RSR_EX.

```
BEGIN_PROC_TRAPS(arm_irq_exception)
        sub     lr,     lr,     #4
        srsdb   r13_svc!
        /* Enter supervisor mode */
        cps     svc_mode
        sub     sp,     sp,     #PT_SIZE-8

        /* save user - banked regs  */
        stmib   sp,     {r0-r14}^
        /* Indicate IRQ to
          soc_handle_interrrupt() */

        mov     r1,     #0


BEGIN_PROC_TRAPS(avr32_irq_exception)
        sub     lr,     4

        sub     sp,     PT_SIZE-8
        /* save user - banked regs  */
        stmts   sp,     r0-r14
        /* Indicate IRQ to
          soc_handle_interrupt() */
        mov     r11,     0
```

## *D. Addressing modes*

Both architectures support direct, indirect and indexed addressing modes. ARM splits indexed mode into pre-indexed and post-indexed, the difference being the use of the modifications on the indexing register. AVR32 replaces this by allowing certain instructions to post-increment or pre-decrement the indexing register, handling the most common cases. AVR32 also has specific instructions for PC-relative and SP-relative loading and storing, instructions that are faster (the result is available after one cycle regardless of data dependencies) and occupy less space (half-word instructions in both cases).

## III. Cache Control

Cache control on the ARM architecture is handled as writes to registers in an on-chip control co-processor. In stark contrast, on AVR32 there is a single dedicated instruction 'CACHE' that can perform all operations (clean, invalidate, flush, lock) on one of four possible caches (on the processor family used in this study, only two caches are available, an L1 instruction cache and an L1 data cache). Cache control is therefore greatly simplified in AVR32. Draining the write buffer is handled by a single instruction as well (sync).

| Op[4:3] | Op[2:0] | Operation | Parameter |
|---|---|---|---|
| 00 | 000 | Flush | Flush mode |
| 00 | 001 | Invalidate | Virtual Address |
| 00 | 010 | Lock | Virtual Address |
| 00 | 011 | Unlock | Virtual Address |
| 00 | 100 | Prefetch | Virtual Address |
| 00 | 101 | Reserved | N/A |
| 00 | 110 | Reserved | N/A |
| 00 | 111 | Reserved | N/A |
| Other | xxx | Reserved | N/A |

For example, the following represents the cache flush of the data cache on ARMv5, using writes to the system co-processor CP15:

```
word_t zero = 0;
__asm__ __volatile__ (
    ERRATA_NOP
    "mcr     p15, 0, %0, c7, c14, 0\n"
    ERRATA_NOP
    ERRATA_NOP
    "mcr     p15, 0, %0, c7, c10, 4\n"
:: "r" (zero)
);
```

On AVR32, as stated, they are reduced to 'cache' instructions:

```
__asm__ __volatile__ (
  "mov    r11, 4              \n"
  "cache  r11[0], 8           \n"
  "sync   0                   \n"
::: "r11"
```

## IV. System Call Convention and Trap Handling

System calls in OKL4 under ARM are made using the swi instruction. The syscall number is not passed as a comment in the swi instruction as expected, instead it is passed in the SP register, while the stack register is saved in the IP (a scratch register). All registers that are not part of the parameter passing convention are saved on the stack, along with the link register (to be popped out into PC after the syscall). This convention has been preserved in our AVR32 port, with the appropriate switch to using registers grouped around R12 for passing parameter and return values.

Furthermore, AVR32 uses the scall instruction for switching to privileged mode, and it does not support any arguments, being a half-word instruction.

We will now present an example of a syscall wrapper for the L4_Mutex syscall, written for AVR32. Certain syscalls require object dereferencing before jumping into privileged mode, this however is an example of the simplest of wrappers, where no such preparation is required. The wrapper saves the registers that are not part of the call (R12 and down are parameters/results), together with the link register, saves the stack and puts the syscall number in the stack register.

```
LABEL(L4_Mutex)
   stm     --sp, {r0-r7, lr}

   mov     r1, sp
   mov     sp, #SYSNUM(mutex)
   scall

   ldm ++sp, {r0-r7, pc}
```

### A. Syscall Trap

On the kernel side syscall handling is very similar, both have exception tables where a syscall-specific entry is found. While we generally followed the ARMv6 implementation, here AVR32 is more similar to ARMv5, in that saving the user status register and program counter is done automatically on enter and restored on exit (scall and rets instructions).

### B. Interrupt Traps

Handling IRQ/FIQ traps is straightforward, the context is saved on the system stack then the platform dependent soc_handle_interrupt is called. The registers saved differ based on what registers are banked, from one processor family to another, as well as from one interrupt level to another (IRQ/INT0 has no banked registers, while FIQ/INT3 has 4).

## V. Memory Management

Both architectures provide a Memory Management Unit responsible for mapping virtual to physical addresses. This translation process uses a TLB cache and a page table system.

The okl4 kernel provides a MMU interface for both v5 and v6 versions of ARM architecture, which are different in terms of page sizes, access permissions and other page control issues. The methods and classes from *pistachio* map the hardware structures and provide page handling functionalities. A port of this code for the avr32 architecture implements the interface in a similar manner, rewriting it at at a structural and functional level. While on ARM the page table organization and its handling are implemented in hardware, on AVR32, these must be implemented at the OS level. Therefore, functionalities not included in OKL4, like page tables management, had to be written. In the next paragraphs we will briefly describe the way we chose to implement these MMU components.

The page tables are organized on two levels as discribed in *Figure 1*, the first one keeping references to second level tables that contain page entries. The virtual address consists of a 10 bit offset in the level 1 table(hence 1024 entries) followed by offsets in the level 2 table and in the page. The pages have four possible sizes 1 kB, 4 kB, 64 kB and 1 MB, complicating the addressing process of the l2 tables: there is no way to know when how many bits should when fetching the L2 offset from the virtual address. The solution we consider is to have multiple entries in the l2 table for the same page. For example a 1 kB page requires just one entry, while a 4Kb page will have 4 entries. In this way, using a 12 bit offset from the virtual address we can match any page entry. The ARM architecture [4] provides a two level organization too, but it maintains page entries on both levels, with a different entry structure for each page type and no bitfield for the page size. For AVR32 we chose a simpler approach, we used only one entry type, having 2 bits assigned for the page size.

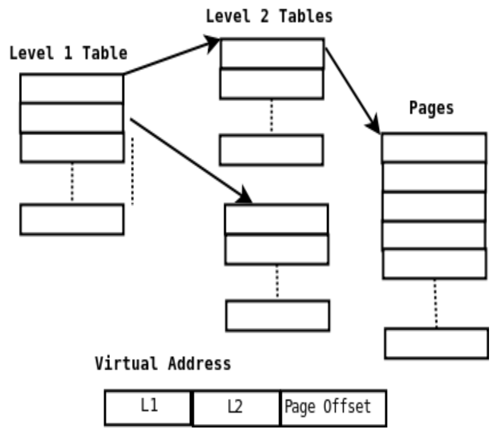

Figure 1. Page Table Structure.

The format of the entries is the one suggested in the architecture document [1], and it is perfectly mapped on the bits of TLBLO (TLB Entry Register Low Part) register. The data from this register is loaded into the TLB using one of the TLB handling instructions.

## VI. Drivers

OKL4 provides a System on Chip Software Development Kit (SoC SDK) that facilitates porting to a new system-on-chip. Besides the API, it contains the necessary components to build a new SoC module that combined with the core kernel generates the final system image. The developer's task is to implement the two header files of the SoC API: *soc/soc.h* and *soc/interface.h*. The first one contains functionality required by a SoC implementation, providing an interface to the hardware platform. The methods of the second header are implemented in the kernel and used by the SoC functions of the first header.

The *soc.h*'s functionality can be categorized as:

- Versioning
- System Start Up
- Interrupt configuration and control
- Timer
- Cache Operations
- Debug support
- System error
- Platform specific

For Atmel's AP7000 it is necessary to implement all these components, except Cache Operations because there are no caches that are not CPU specific.

Having as a model the ARM platforms' SoC implementation, we divided our code into several files, one for each component (eg. *interrupt.c*). In addition to these, we implemented auxiliary data structures and methods necessary for controlling the peripherals through their registers.

Porting the Interrupt configuration routines proved to be easier than for ARM platforms, AP7000's Interrupt Controller allowing control on 4 levels. Therefore, interrupts are grouped into 4 priority levels, and can be masked directly from the status register, and not from interrupt lines registers.

In addition to SoC API implementation, drivers for real time clock and usart(Universal serial asynchronous receiver/transmitter) must be implemented. Even if this seems separate from the rest of the SoC code, in fact these drivers are part of the SoC-specific implementation of OKL4, because timers, the realtime clock and serial interfaces are on-chip. These drivers follow a pattern, each implementing the following functions (presented for a timer driver for platform x),

```
device_setup_impl(
      struct device_interface*,
      struct x_timer *,
      struct resource*)
device_enable_impl(
      struct device_interface*,
      struct x_timer *)
device_disable_impl(
      struct device_interface*,
      struct x_timer *)
```

for setup, enabling and disabling the device. The setup callback deals with allocating memory needed for buffers, while functionality of the enable and disable functions is straightforward. Each driver has also operation-specific callbacks, such as get_tick or set_tick for timers, or do_rx for serial drivers. There is a callback for interrupt handling as well, all the functionality for the peripheral is in this driver, the main code from the SoC code just calls it.

## VII. Compiler Support

While on ARM several compilers are available (and OKL4 is compatible with 3 of them), on AVR32 only a gcc version is available (not an official port yet), which supports inline assembly in the same way that the ARM branch of gcc does.

## VIII. Conclusion

This study has presented the key issues that need to be addressed when porting OKL4, or a microkernel in general, from ARM to AVR32. Unfortunately, this paper could not be accompanied by a functional port of OKL4, and results could not be shown. However, bits and pieces of the core functionalities have been detailed and should be taken as a starting point by the interested reader.

## References

[1] Atmel, "Avr32 architecture document."

[2] ——, "Avr32 ap technical reference manual."

[3] ——, "At32ap7000 preliminary."

[4] ARM, "Arm architecture manual."

[5] ——, "User's manual s3c2410a."

[6] O. K. Labs, "Okl4 microkernel reference manual - api version 03."

[7] ——, "Okl4 soc developers manual."

[8] G. Heiser, "Virtualization for embedded systems."