7 - Embedded systems (2013 version, not really a part of the 2014 course)

TSEA81 - Computer Engineering and Real-time Systems

This document is released - 2013-12-16 - first version (new homepage)

Author - Ola Dahl

Lecture - 7 - Embedded systems (2013 version, not really a part of the 2014 course)

The lecture gives information about embedded systems, through a design and implementation example, where an RTOS and an application shall execute on a target system.

The target system used is Beagleboard[1], and the RTOS used is *Simple_OS*.

The notation [RT] refers to the book Realtidsprogrammering[2].

When describing hardware-dependent features and constraints, the presentation has its main focus on the ARM architecture, which is present in the Beagleboard. Where appropriate, comparisons with the Intel-x86 architecture are presented. For additional information regarding implementation of an RTOS for the Intel architecture, see Chapter 15 in [RT].

[1] http://beagleboard.org/

[2] https://www.studentlitteratur.se/#31445

Hardware

The selected hardware board is Beagleboard[3], of the type Beagleboard-xM[4].

The Beagleboard contains an OMAP DM3730 Multimedia processor[5]. The OMAP DM3730 is a System-on-Chip, *Digital Media Processor*, with an ARM Cortex-A8 processor[6]. In addition, it contains a C64+ Digital Signal Processor[7], a POWER SGX Graphics Accelerator, a Video accelerator, and a set of interfaces, such as USB, McBSP, and I2C. It also contains on-board memory, and interfaces to external memory.

The ARM Cortex-A8 Technical Reference Manual[8] contains detailed information about the ARM Cortex-A8 processor.

Information about the Intel architecture can be found in Intel 64 and IA-32 Architectures Developer's Manual: Combined Volumes[9].

[3] http://beagleboard.org/

[4] http://beagleboard.org/static/BBxMSRM_latest.pdf

[5] http://www.ti.com/product/dm3730

[6] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/index.html

[7] http://www.ti.com/lit/ug/spru871k/spru871k.pdf

[8] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/index.html

[9] http://intel.ly/TdFCFI

Architecture

The ARM architecture[10] is a RISC architecture, with a uniform register file and a load/store architecture where data processing operations act on registers and not on memory.

[10] http://en.wikipedia.org/wiki/ARM_architecture

The evolution of the ARM instruction set is seen in its version numbers. The ARM Cortex-A8[11] has an instruction set with the version label *ARMv7-A*, which is instruction set version 7, profile A.

The current ARM architecture has a single, 32-bit byte-addressable address space. There are however announcements also of a 64-bit ARM Architecture[12].

The Cortex-A8 processor belongs to the A-series of processors (there is also an M series, and an R series), and it has an MMU[13].

The Intel processor is a CISC processor, with instructions of different size and different complexity, and with more instructions operating directly on memory (when compared with a typical RISC architecture).

[11] http://www.arm.com/products/processors/cortex-a/cortex-a8.php

[12] http://bit.ly/WrtC4I

[13] http://en.wikipedia.org/wiki/Memory_management_unit

Registers

The ARM register map[14] has thirteen 32-bit registers, named *r0* to *r12*. In addition, there are three 32-bit registers, named *r13* to *r15*, that have special use. For these registers, it holds that

- *r13*, also referred to as *sp*, is the *stack pointer*
- *r14*, also referred to as *lr*, is the *link register*
- *r15*, also referred to as *pc*, is the program counter

The ARM architecture also provides a processor status register, denoted *cpsr*, and a saved processor status register, denoted *spsr*.

An ARM processor can execute in different modes. One mode is called *user mode*. The other modes, which are denoted *privileged modes*, includes e.g. *supervisor mode*, and interrupt modes such as *IRQ mode* and *FIQ mode*.

The implementation of *Simple_OS* is done so that *Simple_OS* and its application execute in *supervisor mode*, as long as an interrupt does not occur. When an interrupt occurs, the processor will enter *IRQ mode*.

A larger operating system, such as Linux, executes in *user mode* when an application executes, and it would switch to *supervisor mode* when a *system call* is performed. When an interrupt occurs, it would switch to an interrupt mode, most likely *IRQ mode*.

As can be seen in the ARM register map[15], registers *r13* (sp) and *r14* (lr), together with the saved processor status register *spsr*, are *banked* in all but two of the modes. A register which is banked in a certain mode can be regarded as having a mode-specific value, which is accessible as long as the mode is active.

[14] http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/graphics/register_organization_in_arm_state.svg

[15] http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/graphics/register_organization_in_arm_state.svg

Instructions

The instruction set for the ARM Cortex-A8 is described in the ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition - Issue C[16], which can be downloaded in pdf-format after a registration.

[16] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html

In the implementation of *Simple_OS* for ARM, one finds instructions like

- *mov* - copy data
- *ldr* - load
- *str* - store
- *ldmfd* - load multiple data
- *stmfd* - store multiple data
- *add* - addition
- *sub* - subtract
- *bl* - branch and link
- *svc* - supervisor call
- *msr* - move to special register
- *mrs* - move to register from special register
- *srsdb* - store return state (*lr* and *spsr*) of the current mode to a stack of a specified mode
- *cps* - change processor mode

Examples of instructions used in the Intel-x86 implementation of *Simple_OS* are

- *movl* - copy data
- *popl* - pop data from the stack
- *ret* - return from subroutine
- *pushl* - push data onto the stack
- *addl* - addition

The above listed Intel instructions are expressed with names used in the GNU Assembler[17]. Some of these names have, compared with the names used e.g. in the Intel manuals[18], an appended letter *l* (for long, i.e. 32 bit). An example is the instruction *movl*.

[17] http://en.wikipedia.org/wiki/GNU_Assembler

[18] http://intel.ly/TdFCFI

Interrupts

The ARM architecure defines *exceptions* as events which cause the processor to branch to a specific address, while at the same time saving the current processor status and the return address. The address selected for the branch is referred to as *exception vector address*, and it depends on the *type* of the exception. The instruction stored at the exception vector address is typically a branch instruction, which makes the processor execute a user-defined routine, e.g. an interrupt handler.

External interrupts are considered to be one of the exception types. Other exception types are e.g. reset and supervisor call (issued by the instruction *svc*), but also error conditions, such as Undefined Instruction and Prefetch Abort.

The exception vector addresses are located at specific locations in memory. The locations are defined by an *exception base address*. The exception vectors are located at specified offsets relative to the exception base address. As an example, the exception vector for an external interrupt, referred to as an IRQ exception, is located at offset *ox18*. The exception vector for supervisor call, referred to as an SVC exception, is located at offset *oxo8*. Information about these offsets can be found e.g. on page 18 in The ARM Architecture - with a focus on v7A and Cortex-A8[19], but also on this page from the ARM Cortex-R4 Technical Reference Manual[20].

When an exception is entered, the processor status register *cpsr* is saved in the SPSR for the exception mode that is handling the exception.

Upon exit from an exception, the program counter and the cpsr need to be restored, to values enabling the execution to continue at the point where the exception occurred, in a correct way.

[19] http://www.arm.com/files/pdf/ARM_Arch_A8.pdf

[20] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0363g/Babfeega.html

Device communication

The AM/DM37x Multimedia Device - Technical Reference Manual[21], which is part of the documentation for the OMAP DM3730 processor[22], contains useful information for communication with peripheral devices.

Chapter 2 of the AM/DM37x Multimedia Device - Technical Reference Manual[23] contains information about the *memory map*. From this map it is possible to determine the address ranges associated with a certain type of peripheral.

For our purpose it is of interest to use a hardware timer, for generation of periodic interrupts. It is also of interest to use a serial port, for communication with a host computer.

It can be seen, in Section 2.3.2.2 of the above mentioned Techni-

[21] http://www.ti.com/lit/ug/sprugn4r/sprugn4r.pdf

[22] http://www.ti.com/product/dm3730

[23] http://www.ti.com/lit/ug/sprugn4r/sprugn4r.pdf

cal Reference Manual[24], that timer number one has the base adress *ox48318000*.

It can also be seen, in Section 2.3.2.3, that one of the UARTs has the base address *ox49020000*. Data written to this UART will appear on the serial connector of the Beagleboard, and from there it can be sent to a host computer.

[24] http://www.ti.com/lit/ug/sprugn4r/sprugn4r.pdf

Mixing C and assembly

Parts of a real-time operating system need to be implemented in assembly. As has been discussed in Lectures *5 - Real-time kernel* and *6 - Real-time operating system*, assembly code is needed for implementation of *task start* and for *task switch*. Assembly code is used for saving and restoring data to and from the stack of a task, and also for handling a switch between two tasks.

In addition, assembly code is often needed when implementing an interrupt handler.

A task start and a task switch can be done by calling assembly code from C code. In this case, the assembly code must be written so that parameter passing conventions, and conventions for return values are obeyed.

Such conventions are often referred to as *calling conventions*, or *procedure call standards*.

Information about calling conventions for ARM can be found in this presentation, from University of Nottingham[25].

Information about calling conventions for the Intel-x86 architecture can be found in the Wikipedia article on x86 calling conventions[26].

When calling assembly from C, calling conventions must be obeyed when writing the assembly code, taking into account that the compiler generates code for parameter passing and for handling of return values. As an example, for the ARM architecture, parameters are typically passed in registers while for the Intel 32-bit architecture, parameters are typically passed as values stored on the stack. A return value, for both ARM and Intel, is often passed in a register.

The calling conventions must be obeyed also when calling C from assembly, as done e.g. in an interrupt handler. In this case the code handling the parameter transfer, e.g. storing parameter values in registers or on the stack, which is generated by the compiler when calling assembly from C, must be written by the programmer.

It is possible to practically investigate the calling conventions used, by generating assembly code from C code. The generated assembly code can then be inspected, and the calling conventions can be decoded, or verified against available documentation. When using *gcc*, assembly code can be generated by using the switch *-S* to the

[25] http://www.cs.nott.ac.uk/~dfb/G51CSA/Slides/lect08-2011.pdf

[26] http://en.wikipedia.org/wiki/X86_calling_conventions

compiler.

Hello world

As a first step in an RTOS development, and also as a way to get acquainted with a specific hardware platform, a minimal C program, to be executed *without* an operating system, can be constructed.

Such a minimal, "Hello, world"-style program, needs to start when the system is started. It also needs to perform certain hardware initialisations, and it needs to call the entry point for C code, often defined by a function named *main*.

In addition, the program needs to perform initialisation of interrrupt handling. Depending on the processor architecture it also needs to force the processor into a certain mode, e.g. *supervisor mode* for ARM, or protected mode[27] for Intel.

[27] http://en.wikipedia.org/wiki/Protected_mode

A compiler is needed. A compiler, named *Sourcery CodeBench for ARM EABI* and executable on a Linux host, can be obtained from Mentor Graphics[28]. This compiler is used in the course, and the downloaded package also includes other tools, such as a linker.

[28] https://sourcery.mentor.com/sgpp/lite/arm/portal/release1802

Linking of a program, for execution on the Beagleboard, can be done using the linker included in the Sourcery CodeBench for ARM EABI. It is then required to specify the memory layout for the linked program, for example where data shall be stored, and where code shall be stored.

The memory layout for a linked program can be specified in a linker script[29]. A linker script that can be used on the Beagleboard can be constructed by creating a file with contents as

[29] http://www.delorie.com/gnu/docs/binutils/ld_6.html

```
ENTRY(start)
SECTIONS
{
 . = 0x80000000;
 .startup . : { obj/startup_arm_bb.o(.text) }
 .text : { *(.text) }
 .data : { *(.data) }
 .bss : { *(.bss) }
 . = . + 0x5000;
 stack_bottom = .;
}
```

This linker script is used also for linking *Simple_OS* and an application.

The linker script defines a *start address*, in this case the address *ox8ooooooo*. This start adress will be used by the linker so that the object code, when linked, is linked against this address. It is of course

then also important, when the program is downloaded to the Beagleboard, that it is placed in memory at this start address.

The linker script also refers to an object file, named *startup_arm_bb.o*. This object file contains code, which has the task to perform initialisations, and then call *main*. This code, which can be referred to as *startup code*, is here created in a minimal version. It is written in assembly, as

```
.global start
start:
ldr sp, =stack_bottom
bl main
b .
```

For additional information on this type of programming, sometimes referred to as *bare-metal programming*, can be obtained from this article describing a Hello World program on an ARM processor[30]. It can also be obtained from a series of articles in EE Times, about bare-metal ARM systems[31].

[30] http://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/

[31] http://bit.ly/WQFdPE

The C-code for the program, to be compiled and linked, using a linker script and startup code as described above, is given by

```
#include "console.h"

int main(void)
{
    console_put_string("A bare metal C-program!\n");
    return 0;
}
```

Task start - first attempt

A first version of an implementation of *task start* can be done, using instructions which are used when programming with subroutines. When doing this, the calling conventions must be obeyed. This means that for ARM, a subroutine call places the return adress in *lr*, and a subroutine return can be done by copying *lr* to *pc*. For Intel, it means that a subroutine call places the return adress on the stack, and a subroutine return can be done by popping the stack into *pc*.

A function, named *context_restore*, was described in Lecture *5 - Real-time kernel*. Its purpose is to start a task. Its C-prototype is given by

```
/* context_restore: starts task with current
   stack pointer new_stack_pointer */
void context_restore(mem_address new_stack_pointer);
```

An implementation for ARM can be done as

```
context_restore:
        @ copy parameter value to sp
        mov sp, r0
        @ restore registers from stack
        ldmfd sp!, {r0-r12, r14}
        @ restore pc
        ldmfd sp!, {pc}
```

An implementation for the Intel x86 32-bit architecture can be done as

```
context_restore:
        # copy parameter value to esp
        movl 4(%esp), %esp
        # restore registers
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax
        # restore pc
        ret
```

The function *context_restore* copies data from the stack to registers in the procesor. Hence, before *context_restore* is called, the stack must be prepared, so that the appropriate data are stored on the stack. This can be done, in C-code, in a function *prepare_stack*, implemented as

```
void prepare_stack(
    stack_item *stack, int stack_size,
    mem_address *sp,
    void (*task_function)(void),
    int n_saved_registers, int n_bytes_per_register)
{
    int i;
    stack[stack_size-1] = (stack_item) task_function;
    for (i = 0; i < n_saved_registers; i++)
    {
        stack[stack_size-i-2] = 0;
    }
    *sp = (mem_address) &stack[stack_size-1];
    *sp -= n_saved_registers * n_bytes_per_register;
```

```
}
```

and used, e.g. as

```
    prepare_stack(task_1_stack, STACK_SIZE, &task_1_sp,
                  task_1_function,
                  n_saved_registers, n_bytes_per_register);

    context_restore(task_1_sp);
```

Task switch - first attempt

The mechanisms used for subroutines can also be used, in a first attempt, for implementation of *task switch*. During a task switch, the program counter of the task to be suspended is saved, and the program counter of the task to be resumed is restored. In addition, there must be a switch from the stack of the task to be suspended to the stack of the task to be resumed.

When implementing task switch using mechanisms for subroutines, the program counter can be *saved* in the same way as when a subroutine is *called*, and it can be *restored* in the same way as in a subroutine *return*.

A function, named *context_switch*, was described in Lecture *6 - Real-time operating system*. Its purpose is to perform a context switch. Its C-prototype is given by

```
/* context_switch: performs a task switch, by
   saving registers, and storing the saved value
   of the stack pointer in old_stack_pointer,
   and then restoring registers from the stack
   addressed by new_stack_pointer */
void context_switch(mem_address *old_stack_pointer,
                    mem_address new_stack_pointer);
```

An implementation for ARM can be done as

```
context_switch:
        @ save link register
        stmfd sp!, {lr}
        @ save registers
        stmfd sp!, {r0-r12, r14}
        @ copy sp to address referred to by
        @ first parameter
        str sp, [r0]
        @ switch stack, copying new stack
        @ pointer in second parameter to sp
```

```
        mov sp, r1
        @ restore registers
        ldmfd sp!, {r0-r12, r14}
        @ restore pc
        ldmfd sp!, {pc}
```

An implementation for the Intel x86 32-bit architecture can be done as

```
context_switch:
        # save registers
        pushl %eax
        pushl %ebx
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        # save esp
        movl %esp, %ebp
        # rewind stack, to find first parameter
        addl $28, %ebp
        # store first parameter in eax
        movl 4(%ebp), %eax
        # copy esp to adress referred to by
        # first parameter
        movl %esp, (%eax)
        # switch stack, copying new stack
        # pointer in second parameter to esp
        movl 8(%ebp), %esp
        # restore registers
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax
        # restore pc
        ret
```

The function *context_switch* is called from a function called *task_switch,* as

```
        context_switch(&task_1_sp, task_2_sp);
```

Interrupts

A real-time operating system uses interrupts. A periodic interrupt is used as a time-base, and there may be other interrupts, which are triggered when external events occur.

An interrupt makes the processor execute an interrupt handler. As a preparation step, before this can occur, the interrupt handling hardware must be initialised, and addresses to interrupt handlers must be stored so that they can be used when interrupts occur.

In a Beagleboard, interrupts can be initialised by first defining a symbol, representing an address where a jump instruction is stored, as

```
int_vector:
        ldr pc, [pc, #24]
```

Then, an interrupt handler for the ARM *IRQ* interrupt can be installed, by first storing the jump instruction at the exception offset corresponding to *IRQ*, and then storing the address of the interrupt handler at the address to which the jump instruction jumps. This can be done by a subroutine, called *setup_int_handler*, implemented as

```
setup_int_handler:
        ldr r0, =0x4020FFDC
        ldr r1, =int_vector
        ldr r2, [r1]
        str r2, [r0]
        ldr r0, =0x4020FFFC
        ldr r1, =int_handler
        str r1, [r0]
        mov pc, lr
```

An actual interrupt handler can be implemented in assembly, as

```
int_handler:
        @ adjust lr (needed in IRQ)
        sub lr,lr,#4
        @ store lr and spsr on supervisor mode stack
        srsdb   #MODE_SUPERVISOR!
        @ change mode to supervisor mode
        cps     #MODE_SUPERVISOR
        @ save registers
        stmfd sp!, {r0-r12}

        @ clear interrupt (timer)
        ldr r0, =0x48318018
        ldr r1, =0x00000011
```

```
        str r1, [r0]
        ldr r0, =0x48318018
        ldr r1, [r0]

        @ call handler
        bl int_handler_function

        @ acknowledge interrupt
        ldr r3,=0x48200048
        mov r1,#1
        str r1,[r3]

        @ restore registers
        ldmfd sp!, {r0-r12}
        @ restore lr
        ldmfd sp!, {lr}
        @ adjust sp
        add sp, sp, #4
        @ restore pc, and restore cpsr from spsr
        movs pc, lr
```

When an IRQ interrupt occurs on an ARM processor, the processor switches mode to *IRQ mode*. In the interrupt handler above, the mode is changed from IRQ mode to *supervisor mode*. This is an implementation decision, and in this example (implementing *Simple_OS* on ARM), it simplifies the handling of (hardware) stacks, since only one hardware stack is used (the supervisor mode stack).

The interrupt handler above calls a subroutine, named *int_handler_function*. This subroutine, in turn, calls a function implemented in C. In the *Simple_OS* implementation, the *tick_handler_function* in the *tick_handler* module is called.

A C-program, that can be used for testing the implementation of periodic interrupts, can be implemented as

```
int main(void)
{
    DISABLE_INTERRUPTS;

    console_put_string("timer_interrupt\n");

    tick_handler_init();

    setup_int_handler();
    enable_timer_interrupts();
```

```
    ENABLE_INTERRUPTS;
    return 0;
}
```

Interrupts and task stack layout

When an interrupt occurs, *cpsr* and *pc* are saved, *by the hardware*.

Saved values of *cpsr* and *pc* must be restored when execution shall be resumed.

If, during an interrupt, it is decided, in the interrupt handler or in code called from the interrupt handler, that no context switch shall occur, then instructions for return from interrupt can be used. These instructions restore the saved *cpsr* and the saved *pc*

If, on the other hand, it is decided, again during an interrupt, that a context switch *shall* occur, it must be ensured that the saved values of *cpsr* and *pc* are saved on the *stack of the running task*, i.e. the task that was interrupted. In addition, corresponding values for these registers need to be restored from the task that shall be resumed.

Our initial attempt, using "thinking in subroutines", used a stack layout with *pc* and registers saved. This approach could be sufficient if all tasks switches were intiated by tasks.

As seen by the above reasoning, if task switches also can be initiated by interrupt handlers, this requires a stack layout where also *cpsr* is stored.

This can be implemented, by changing the stack layout, in all places in the RTOS implementation where it is used, explicitly or implicitly, so that all tasks not running have the *same layout* of the data stored on their stacks.

This modification of the stack layout also affects task creation, and task start, since the stack must be prepared according to the modified stack layout.

It must also be taken into account when performing a task-initated task switch. The reason here is that it is not possible to know, during a task-initated task switch, if the task to be resumed was previously suspended by a task-initiated task switch or an interrupt-initiated task switch. It is also not possible to know if the task to be resumed has executed, or if this is the first time the task runs.

Task start using return from interrupt

The function for starting a task, with C-prototype according to

```
/* context_restore: starts task with current
   stack pointer new_stack_pointer */
```

```
void context_restore(mem_address new_stack_pointer);
```

can be modified, using the same type of instructions as when returning from interrupt. The modified implementation, for ARM, is given by

```
context_restore:
        @ copy parameter value to sp
        mov sp, r0
        @ restore registers from stack
        ldmfd sp!, {r0-r12, r14}
        @ save r0 on stack
        stmfd sp!, {r0}
        @ load r0 with cpsr for task
        ldr r0, [sp, #4]
        @ store r0 value in spsr
        msr spsr, r0
        @ restore r0 from stack
        ldmfd sp!, {r0}
        @ make sp refer to start address of task
        add sp, sp, #4
        @ restore pc, and restore cpsr from spsr
        ldmfd sp!, {pc}^
```

As can be seen in the above listing, the instruction `ldmfd^` is used. This instruction is similar to `ldmfd` in that it loads values, in this case a value is loaded to the program counter *pc*, but in addition it also loads *cpsr* from the saved processor register *spsr*.

The preparation of the stack, during task creation must be modified, so that a predefined value to be stored in the processor status register *cpsr* is stored on the stack. This predefined value can, for an ARM processor, be chosen so that the processor is executing in supervisor mode and the interrupts are enabled. C-code for performing this preparation is implemented in *Simple_OS*, in the file *task.c*, as

```
static void prepare_stack(
    stack_item *stack, mem_address *sp,
    void (*task_function)(void),
    int n_saved_registers, int n_bytes_per_register)
{
    int i;
    stack_item *stack_ref;
    stack_ref = stack;
    *stack_ref = (stack_item) task_function;
    stack_ref--;
    /* store a value of cpsr, interrupts
```

```
       enabled, supervisor mode */
    *stack_ref = 0x00000013;
    for (i = 0; i < n_saved_registers; i++)
    {
        stack_ref--;
        *stack_ref = 0;
    }
    *sp = (mem_address) stack;
    *sp -= 4;
    *sp -= n_saved_registers * n_bytes_per_register;
}
```

The function *task_start*, in *task.c*, implements the task start, by calling *context_restore*, as

```
void task_start(int task_id)
{
    /* a pointer to a TCB */
    task_control_block *tcb_ref;

    /* get a pointer to the TCB associated with
       task identity task_id */
    tcb_ref = tcb_storage_get_tcb_ref(task_id);

    /* set Task_Id_Running to task id of new task */
    Task_Id_Running = task_id;

    /* restore context for the task with this
       TCB */
    context_restore(tcb_ref->stack_pointer);
}
```

Task switch using software interrupt

In an ARM processor, a software interrupt (SVC) saves *cpsr* and *pc*. This is done in the same way as when an external interrupt occurs. This is expected, since a software interrupt, as well as an external interrupt, are different kinds of *exceptions*.

If a task-initiated task-switch is implemented using a software interrupt, the resulting stack layout - with respect to the saved values of *cpsr* and *pc* - will therefore be the same as when an interrupt occurs. This makes it advantageous to use a software interrupt instead of a subroutine call to implement a task-initiated task switch.

A subroutine, called *context_switch_swi* can be created. This subroutine performs a software interrupt, using the *svc* instruction, and

it can be implemented as

```
context_switch_swi:
        @ save lr on stack
        stmfd sp!, {lr}
        @ do the software interrupt
        svc 0x10
        @ restore lr
        ldmfd sp!, {lr}
        @ return to caller
        bx lr
```

The subroutine *context_switch_swi* can be called from C, and its C-function prototype becomes

```
/* context_switch_swi: performs a task switch, using software interrupt,
   by saving registers, and
   storing the saved value of the stack pointer in old_stack_pointer, and
   then restoring registers from the stack addressed by new_stack_pointer */
void context_switch_swi(mem_address *old_stack_pointer, mem_address new_stack_pointer);
```

This assumes that the actual function doing the context switch has been installed as an exception handler for software interrupts. This function is implemented in assembly, as a subroutine named *context_switch,* as

```
context_switch:
        @ save link register
        stmfd sp!, {lr}
        @ save r0 on stack, on a free place
        str r0, [sp, #-8]
        @ store spsr in r0
        mrs r0, spsr
        @ save r0 on stack
        stmfd sp!, {r0}
        @ restore previously saved r0
        ldr r0, [sp, #-4]
        @ save registers
        stmfd sp!, {r0-r12, r14}
        @ copy sp to address referred to by
        @ first parameter
        str sp, [r0]
        @ switch stack, copying new stack
        @ pointer in second parameter to sp
        mov sp, r1
        @ restore registers
        ldmfd sp!, {r0-r12, r14}
```

```
@ save r0 on stack
stmfd sp!, {r0}
@ load r0 with cpsr for task
ldr r0, [sp, #4]
@ store r0 value in spsr
msr spsr, r0
@ restore r0 from stack
ldmfd sp!, {r0}
@ make sp refer to start address of task
add sp, sp, #4
@ restore pc, and restore cpsr from spsr
ldmfd sp!, {pc}^
```

A task-initiated context switch can now be initiated, from C-code. This is done, in *Simple_OS*, in the function *task_switch* in *task.c*, as

```
context_switch_swi(old_stack_pointer, new_stack_pointer);
```

Task switch from interrupt

When an interrupt occurs, registers must be saved. If these registers are saved on the stack of the running task, parts of a context switch (if this is decided, during the interrupt), is already done.

The routine for context switch, as described above, saves registers on the stack of the calling task. If this routine should be re-used for a task switch which is initiated from an interrupt, this would lead to a scenario where the registers are saved twice.

Instead, a different routine, named *context_switch_int* can be implemented. This routine shall be called when a *task switch initiated from interrupt* shall occur.

In order to save a reference to the saved registers, i.e. the registers that were saved when the interrupt occured, the interrupt handler can save the value of the stack pointer, when it refers to the saved registers.

The saved stack pointer will then be used in *context_switch_int*, where it will be copied to the TCB of the *task to be suspended*. Then, the stack of the *task to be resumed* will be located, as is done during task switch initiated from a task, and registers and program counter will be restored, from this stack.

The subroutine *context_switch_int* can be called from C, and its C-prototype is

```
/* context_switch_int: performs a task switch, from
   interrupt, by storing a saved value of the stack
   pointer in old_stack_pointer, and then restoring
   registers from the stack addressed by
```

```
   new_stack_pointer */
void context_switch_int(
    mem_address *old_stack_pointer,
    mem_address new_stack_pointer);
```

It is implemented, in assembly, as

```
context_switch_int:
        @ save lr
        stmfd sp!, {lr}
        @ save r3
        stmfd sp!, {r3}
        @ save r0 and r1
        stmfd sp!, {r0, r1}
        @ get stack pointer saved by interrupt handler
        bl int_status_get_saved_stack_pointer
        @ store this stack pointer in r3
        mov r3, r0
        @ restore r0 and r1
        ldmfd sp!, {r0, r1}
        @ copy r3 to address referred to by
        @ first parameter
        str r3, [r0]
        @ restore r3
        ldmfd sp!, {r3}

        @ clear interrupt active flag
        bl int_status_clear_interrupt_active

        @ restore lr
        ldmfd sp!, {lr}

        @ switch stack, copying new stack
        @ pointer in second parameter to sp
        mov sp, r1
        @ restore registers
        ldmfd sp!, {r0-r12, r14}
        @ save r0 on stack
        stmfd sp!, {r0}
        @ load r0 with cpsr for task
        ldr r0, [sp, #4]
        @ store r0 value in spsr
        msr spsr, r0
        @ restore r0 from stack
        ldmfd sp!, {r0}
```

```
@ make sp refer to start address of task
add sp, sp, #4
@ restore pc, and restore cpsr from spsr
ldmfd sp!, {pc}^
```

Iterations in RTOS development

The description in this Lecture provides a way to implement a small real-time operating system, using an iterative method.

As a first step, the hardware is investigated. This is described in Section **??**. Then, a "Hello, world"-program is implemented, and executed directly on the hardware as described in Section **??**.

Using methods for calling, and return from, subroutine, a first version of *task start*, as described in Section **??**, and *task switch*, as described in Section **??**, can be implemented.

Interrupts are then investigated, and a program which is interrupted by a periodic interrupt can be implemented, as described in Section **??**.

As a consequence of the hardware mechanisms used for interrupts, it is seen that the processor status register needs to be stored on the stack of a task which is not executing. This leads to a modification of the stack layout, as described in Section **??**, so that it also includes the processor status register.

Using the modified stack layout the implementation of *task start* and *task creation* can be modified, as described in Section **??**.

Using the mechanisms for *software interrupt*, the routine for task-initiated context switch can be modified so that it takes into account the modified stack layout. This is described in Section **??**.

An interrupt-initiated context switch can now be implemented, by taking into account that registers are already saved by the interrupt handler. This leads to a separate assembly routine for context swith from interrupt, as described in Section **??**.