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

TSEA81

Computer Engineering and Real-time Systems

Linköping University
Sweden

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

Author - Ola Dahl

Embedded systems

How to make an RTOS run on a Beagleboard

- Hardware - processor, board, peripherals
- Hello world on bare metal - a simple program, without RTOS
- Task start and task switch - thinking in terms of subroutines
- Interrupts and its implications on mechanisms for task switch
- Task switch using software interrupt, and task switch from interrupt

Hardware

- Beagleboard
- ARM Cortex-A8
- OMAP DM3730 Multimedia processor
- Host computer - Linux

ARM registers

ARM has 13 32-bit registers, *r0* to *r12*. Three 32-bit registers *r13*-*r15*, that have special use, since

- *r13* is *sp* - the *stack pointer*
- *r14* is *lr* - the *link register*
- *r15* is *pc* - the program counter

ARM also has the processor status register *cpsr*, and the saved processor status register *spsr*

Processor modes

An ARM processor can execute in different modes.

- User mode
- Privileged modes - e.g. supervisor, IRQ, FIQ
- Some registers have individual copies per mode, e.g. *sp* (r13) and *lr* (r14)

We use supervisor mode, and temporarily also IRQ mode.

ARM instructions

Instructions used, are

- *mov* - copy data
- *ldr* - load
- *str* - store
- *ldmfd* - load multiple data
- *stmfd* - store multiple data
- *add* - addition
- *sub* - subtract
- *bl* - branch and link

ARM instructions

Instructions used, are

- *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

ARM exceptions

- Exceptions - interrupts, supervisor call, undefined instruction, memory system abort, etc.
- Exception vectors stored in memory
- An exception makes the processor branch to an *exception vector address*
- Exception vectors at specified locations
- Exception vector offsets, e.g. *0x18* for IRQ and *0x08* for supervisor call (SVC)
- Exception base adress is default *0x0*, but can be changed.

ARM exceptions

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 enable correct execution at the point where the exception occurred.

Exit from exception can be done using different instructions, e.g. *ldmfd*, with a ^ appended.

Device communication

Where are the peripherals?

- Look for memory map in documentation
- TIMER 1 has its base address at *0x48318000*
- UART 3 has its base address at *0x49020000*

Timer and UART, enables clock interrupts and serial communication to host.

Mixing C and assembly

Calling assembly from C

- Compiler prepares parameter passing and return values
- ARM - parameters in registers and return value in register (mostly)
- Intel - parameters on stack and return value in register (mostly)
- Investigate using -*S* switch to (arm-) gcc.

Calling C from assembly

- Assembly prepares parameter passing and takes care of return values

Hello world on bare metal

How can we make a minimal C-program?

- Compiling and linking
- File format
- Load to target
- Startup - *who is calling main?*

A linker script defines the memory layout

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

Use -*T* switch to *gcc*.

Startup code (in assembly) calls main

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

In a larger system, startup code does a lot more (e.g. initialising data, copying from flash to RAM, sets up hardware etc.)

Hello world minimum version, can run without an OS,

```
#include "console.h"

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

Task start - thinking in subroutines

Use instructions also used when programming with subroutines

- ARM - subroutine call with return adress in *lr*, subroutine return by copying *lr* to *pc*
- Intel - subroutine call with return adress on stack, subroutine return by popping stack into *pc*

The function *context_restore* shall start a task. Its C-prototype is

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

ARM implementation of *context_restore* (first attempt)

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

Task switch - thinking in subroutines

Use instructions as when working with subroutines

- Save program counter as done in subroutine call
- Restore program counter as done in subroutine return

The function *context_switch* shall perform a task switch. Its C-prototype is

```
/* 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);
```

ARM implementation of *context_switch* (first attempt)

```
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}
```

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

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

Using interrupts

- Interrupt initialisation
- Interrupt handler
- C and assembly
- Enable and disable interrupts

How to make a minimum program using interrupts?

Make exception vector adress contain a jump instruction, as

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

Create setup routine in assembly, 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
```

Actual interrupt handler (part one)

```
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]
```

Actual interrupt handler (part two)

```
@ 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
```

The hello world with interrupts C-program becomes

```
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 *no* context switch shall occur, then return from interrupt can be used, since it restores the saved *cpsr* and the saved *pc*
- If context switch *shall* occur, we need to ensure 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.

A modified stack layout

- Our initial attempt, using "thinking in subroutines", used a stack layout with *pc* and registers saved.
- Interrupt handling needs a stack layout where also *cpsr* is stored.
- Change stack layout, in all places, so that all tasks not running have the *same layout* of the data stored on their stacks
- This affects also task creation, and task start, since the stack must be prepared in the correct way

Task start using return from interrupt

Use instructions also used when returning from interrupt when starting a task (instead of return from subroutine)

The implementation of *context_restore* changes to (parts are shown)

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

Task switch using software interrupt

- A software interrupt (SVC) saves *cpsr* and *pc*, in a similar way as is done when an interrupt occurs (since both are exceptions)
- Therefore, use SVC instead of a subroutine call to *initiate* a task switch
- This holds for task switches *initiated from tasks*

First, create a *subroutine*, that does the *SVC*, 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
```

Then, implement the exception handler, i.e. the function that *does* the context switch, as (parts shown)

```
context_switch:
        @ ...
        @ store spsr in r0
        mrs r0, spsr
        @ 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 pc, and restore cpsr from spsr
        ldmfd sp!, {pc}^
```

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
- Context switch routine as before saves registers, so cannot be used directly.
- Create new routine, *context_switch_int*, to be called when a *task switch initiated from interrupt* shall occur.

Task switch from interrupt

- The interrupt handler needs to save the stack pointer, when it refers to the saved registers
- The saved stack pointer will be used by *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 function *context_switch_int* shall perform a task switch from interrupt. 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);
```

and it is implemented in assembly.

Iterative RTOS development

- Hello world on bare metal
- Task start and task switch using return from subroutine
- Interrupts (Hello world with timing)
- Add saved processor status word to common stack layout
- Modify stack preparation - task_start
- Start task using return from exception (return from interrupt) - context_restore
- Task-initiated context switch using Software interrupt (SVC - supervisor call) and return from exception (return from interrupt)
- Interrupt-initiated context switch taking into account that registers already saved by interrupt handler - resume new task as when doing task-initiated context switch