ECE3411 – Fall 2015

Lab 6b.

Context Switching & Task Scheduling

Marten van Dijk, Syed Kamran Haider
Department of Electrical & Computer Engineering
University of Connecticut
Email: {vandijk, syed.haider}@engr.uconn.edu

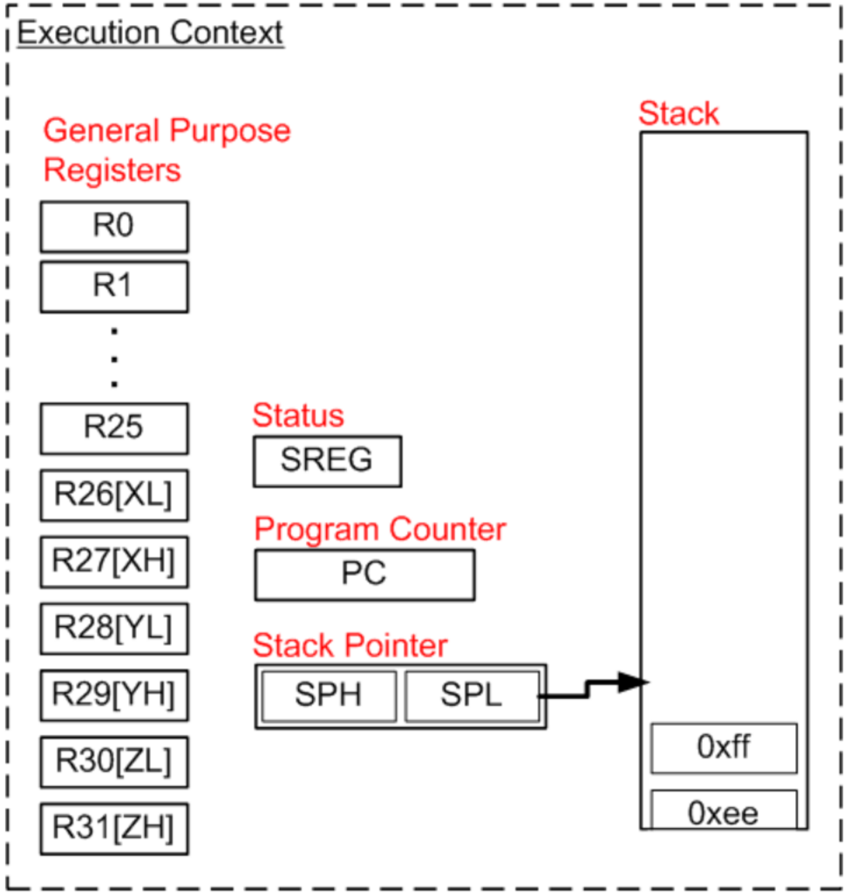
UCONN

With the help of:

www.wikipedia.org
www.freertos.org

Execution Context

- As a task executes it utilizes the processor registers and accesses RAM.
- These resources together comprise the task execution **context**. In particular;
  - The Program Counter (PC)
  - The Status Register (SREG)
  - Processor’s general purpose registers (R0 - R31)
  - The Stack Pointer



Saving the Context

- Each task has its own stack memory area. So the context can be saved by simply pushing processor registers onto the task stack.

```
#define portSAVE_CONTEXT()                     \
asm volatile (                                 \
"push  r0                                      \n\t" \
"in    r0, __SREG__                            \n\t" \
"cli                                           \n\t" \
"push  r0                                      \n\t" \
"push  r1                                      \n\t" \
"clr   r1                                      \n\t" \
"push  r2                                      \n\t" \
"push  r3                                      \n\t" \
"push  r4                                      \n\t" \
…
…
"push  r29                                     \n\t" \
"push  r30                                     \n\t" \
"push  r31                                     \n\t" \
"lds   r26, Current_SP_ptr                     \n\t" \
"lds   r27, Current_SP_ptr + 1                 \n\t" \
"in    r0, __SP_L__                            \n\t" \
"st    x+, r0                                  \n\t" \
"in    r0, __SP_H__                            \n\t" \
"st    x+, r0                                  \n\t" \
);
```

Saving the Context

Referring to the source code on the last slide:

- Processor register R0 is saved first as it is used when the status register is saved, and must be saved with its original value.
- The status register is moved into R0 (2) so it can be saved onto the stack (4).
- Processor interrupts are disabled (3). If portSAVE_CONTEXT() was only called from within an ISR there would be no need to explicitly disable interrupts as the AVR will have already done so. As the portSAVE_CONTEXT() macro is also used outside of interrupt service routines (when a task suspends itself) interrupts must be explicitly cleared as early as possible.
- The code generated by the compiler from the ISR C source code assumes R1 is set to zero. The original value of R1 is saved (5) before R1 is cleared (6).
- Between (7) and (8) all remaining processor registers are saved in numerical order.
- The stack of the task being suspended now contains a copy of the tasks execution context. The kernel stores the tasks stack pointer so the context can be retrieved and restored when the task is resumed. The X processor register is loaded with the address to which the stack pointer is to be saved (8 and 9).
- The stack pointer is saved, first the low byte (10 and 11), then the high nibble (12 and 13).

Restoring the Context

- The context of the task being resumed was previously stored in the tasks stack.
- The kernel retrieves the stack pointer for the task then POPs the context back into the correct processor registers.

```
#define portRESTORE_CONTEXT()                          \
asm volatile (                                         \
"lds  r26, Current_SP_ptr                              \n\t" \
"lds  r27, Current_SP_ptr + 1                          \n\t" \
"ld   r28, x+                                          \n\t" \
"out  __SP_L__, r28                                    \n\t" \
"ld   r29, x+                                          \n\t" \
"out  __SP_H__, r29                                    \n\t" \
"pop  r31                                              \n\t" \
"pop  r30                                              \n\t" \
"pop  r29                                              \n\t" \
"pop  r28                                              \n\t" \
"pop  r27                                              \n\t" \
…
…
"pop  r4                                               \n\t" \
"pop  r3                                               \n\t" \
"pop  r2                                               \n\t" \
"pop  r1                                               \n\t" \
"pop  r0                                               \n\t" \
"out  __SREG__, r0                                     \n\t" \
"pop  r0                                               \n\t" \
);
```

Timer1 ISR: The Scheduler Task

```
/* Interrupt service routine for the RTOS tick. */
ISR( TIMER1_COMPA_vect, ISR_NAKED )
{
        /* This is a naked ISR so the context is saved. */
        portSAVE_CONTEXT();
        /* Store the current Stack Pointer in TCB_Array to retrieve it later */
        TCB_Array[_current_task].stack_pointer = Current_SP;

        /* Switch to Kernel's Stack for Scheduling Computation */
        Current_SP = _kernel_TCB.stack_pointer;
        portSET_SP();

        /* Call the tick function. */
        vPortYieldFromTick();

        /* Store Kernel's latest Stack pointer */
        portREAD_SP();
        _kernel_TCB.stack_pointer = Current_SP;

        /* Retrieve the Stack Pointer of Next task */
        Current_SP = TCB_Array[_current_task].stack_pointer;

        /* Restore the context.  If a context switch has occurred this will restore the context of the task being resumed. */
        portRESTORE_CONTEXT();

        /* Return from the interrupt.  If a context switch has occurred this will return to a different task. */
        asm volatile ( "reti" );
}
```

Helper Scheduler Functions

```
void vPortYieldFromTick( void )
{
   /* Increment the tick count and check to see if the new tick value has caused a delay period to expire.
      This function call can cause a task to become ready to run. */
   vTaskIncrementTick();

   /* See if a context switch is required. Switch to the context of a task made ready to run by
      vTaskIncrementTick() if it has a priority higher than the interrupted task. */
   vTaskSwitchContext();
}
```

```
void vTaskSwitchContext()
{
            /********* Scheduling Policy **********/
            // Round Robin Scheduling
            Scheduler_Round_Robin();

            /***************************************/
}
```

Initializing the Kernel & Registering Tasks

```
int main(void)
{
        initialize_all();

        /* Initialize the Kernel here */
        /* void Initialize_kernel(int num_tasks, double tick_resolution) */
        /* Arguments:
                num_tasks: Max number of tasks you want to schedule
                tick_resolution: Time quantum (in sec) after which scheduling policy should trigger
        */
        Initialize_kernel(2, SCHEDULING_QUANTUM);

        /* Register Tasks */
        /* void RegisterTask(double task_period, void* task_function) */
        /* Arguments:
                task_period: Task Period (in secs). 0 for non-periodic tasks
                task_function: Pointer to the task's function
        */
        RegisterTask(1, (void*) &task1);
        RegisterTask(1, (void*) &task2);

        /* Function to starts the tasks. This gives control to the scheduler. */
        Run_tasks();
        while(1);        /* This loop is never entered anyway. */
}
```

Task1: Round Robin Scheduling

Download the folder W12_Lab1_files.zip.

A simple OS kernel is provided to you which implements Round Robin Scheduling.

You have the following tasks to do:

- Initialize the kernel in the given project with different values of SCHEDULING_QUANTUM and observe the behavior of the output printed on UART.
- Change the delay inside task1() and task2() and observe the behavior of the output printed on UART.
- Why do you sometimes see “Task1” and/or “Task2” being misprinted?

Task2: Rate Monotonic Scheduling

Implement a new function called Scheduler_Rate_Monotonic() in kernel.h that implements Rate Monotonic Scheduling.

- Remove the while(1) loop from task1() and task2() such that both these functions print “Task1”/”Task2” just once.
- Register task1 with a period of 100ms.
- Register task1 with a period of 300ms.
- Set SCHEDULING_QUANTUM to be 50ms.
- In vTaskSwitchContext() function, replace Scheduler_Round_Robin() with Scheduler_Rate_Monotonic()
- How does the output printed on UART look like now?
- What happens if you register more tasks with periods lower than 100ms?