Implementation of a multithreaded environement for AVR using Nuptse.

Julien TOUS

November 13, 2007

1 introduction

This document describes the architecture and internals of an implementation of a multithreaded environement for AVR. This implementation only uses standard Kortex components, without customized intervention of the Nutpse compiler (No architctural aspects). For such an implementation have a look at the Buzz aspect at `svn.forge.objectweb.org/svnroot/think/experiments/buzz`
Large part of this document, and the code it describes is inspired by : `http://www.avrfreaks.net/modules/FreaksArticles/files/14/Multitasking%20on%20an%20AVR.pdf` which itself is greatly inspired by FreeRTOS AVR port. Both are worth taking a look at for a general understanding. The avr-libc doc and FAQ `http://www.nongnu.org/avr-libc/user-manual/FAQ.html` is also a great source of answer for many avr related questions. We will here, mainly focus on the kortex implementation.
As the treatment of interruption is by some way related to multithreaded programming we will also have a look at it and the conveniences given by the avr-libc.

2 AVR specifique low level code

2.1 AVR registers

AVR microcontrollers get 32 general purpose registers (`r0` to `r31`). All those registers are 8 bit wide, althought some instructions treate `X` (`r26-r27`), `Y` (`r28-r29`) and `Z` (`r30-r31`) as 16 bits words. Then `SREG` register contains interruption related informations, `SP` the stack pointer, `PC` the program counter. The state of all those registers, defines a context.
While debuging with gdb, you can monitor those registers with the command :
`(gdb) info reg`
which will output something similar as the following:

```
r0             0x65     101     'e'
```

```
r1      0x0      0        '\0'
r2      0x0      0        '\0'
r3      0x0      0        '\0'
r4      0x0      0        '\0'
r5      0x0      0        '\0'
r6      0x0      0        '\0'
r7      0x0      0        '\0'
r8      0x0      0        '\0'
r9      0x0      0        '\0'
r10     0x0      0        '\0'
r11     0x0      0        '\0'
r12     0x0      0        '\0'
r13     0x0      0        '\0'
r14     0x0      0        '\0'
r15     0x0      0        '\0'
r16     0x0      0        '\0'
r17     0x1a     26       '\032'
r18     0x14     20       '\024'
r19     0x2      2        '\002'
r20     0x0      0        '\0'
r21     0x6c     108      'l'
r22     0x46     70       'F'
r23     0x6      6        '\006'
r24     0x0      0        '\0'
r25     0x2e     2        '\002'
r26     0x1b     27       '\033'
r27     0x2      2        '\002'
r28     0x97     151      '\227'
r29     0x21     33       '!'
r30     0x4d     77       'M'
r31     0x6      6        '\006'
SREG    0x80     128      '\200'
SP      0x802194 0x802194
PC      0x6c96   27798
```

You can also monitor each of them separatly with for example :

```
(gdb) print /x $r1
 $4 = 0xa0
```

`SREG` is mapped as a register at `0x3f` (it is also mapped in memory at `0x5f`). `SPL` and `SPH` (which respectivly contains light weight Byte and heavy weight Byte of `SP`) are mapped as registers at `0x3d` and `0x3e` (they are also mapped in memory at `0x5d` and `0x5e`).

2.2 The trap component.

All context manipulation (Saving, restoring initializing) are handled by the `trap` component. The `trap` component itself uses the macros defined in the `context.h` header.

2.2.1 Switching from a running thread to a suspent one

Assuming a kernel initialize and uses two threads (`thread0` and `thread1`). At a given instant `thread0` is active and ask (we will later see how) to pause it's execution in favor of `thread1`. (table 1) The `switchContext` method from `trap` interface of `trap` component will be call :

```
__attribute__ ((naked)) void SRV_trap__switchContext( jbyte* old_context, jbyte* new_c
    DEBUG_PRINTF("SRV_trap__switchContext \n");
    DEBUG_PRINTF("Saving old_context : %x \n", old_context);
    // Make the global variable context
    // point on the current stack pointer
    // to use it the assembly
    // SAVE_CONTEXT macro.
    context = old_context;
    // Save current context
    SAVE_CONTEXT();

    // restore the new context only if it exists
    if( new_context != 0 ) {
        DEBUG_PRINTF("Restoring new_context : %x \n", new_context);
        // Make the global variable context
        // point on the new stack pointer
        // to use it the assembly
        // RESTORE_CONTEXT macro.
        context = new_context;
        // Set the new context
        RESTORE_CONTEXT();
    }
    else {
        DEBUG_PRINTF("Trying to load a non existing context ? \n");
    }

    // naked function. need manual ret
    __asm__ __volatile__ (  "ret" );
}
```

`switchContext` takes two parameters which should respectively be the pointer to the (current) thread0 context pointer (old_context), and the pointer to the (to be activated) thread1 context pointer (new_context).

`switchContext` uses the `naked` attribute. This prevent the compiler (don't know if any compiler but gcc uses such attributes) from adding data to the stack when jumping to the `switchContext` code : Only the Program Counter (3 bytes on an atm2561) will be added to the stack. (table 2)
Using `DEBUG_PRINTF` (if PRINTF is defined) will then add data on the stack (table 3), and remove it while returning. (table 4) Setting the value of the global variable `context` to `old_context` leaves the stack untouched. We can now execute the assembly code in the `SAVE_CONTEXT()` macro :

```
#define SAVE_CONTEXT()                                  \
asm volatile ( \
"push   r0                  \n\t"   /* Save r0 content on the stack */ \
"in     r0, __SREG__        \n\t"   /* Put SREG content in r0 */  \
"cli                        \n\t"   /* Disable interruptions */ \
"push   r0                  \n\t"   /* Save r0 content (initial SREG value) on the
"push   r1                  \n\t"   /* Save r1 content on the stack */\
"clr    __zero_reg__        \n\t"   /* Put 0 in r1*/\
"push   r2                  \n\t"  /* Save r2 content on the stack */ \
"push   r3                  \n\t"   \
"push   r4                  \n\t"   \
"push   r5                  \n\t"   \
"push   r6                  \n\t"   \
"push   r7                  \n\t"   \
"push   r8                  \n\t"   \
"push   r9                  \n\t"   \
"push   r10                 \n\t"   \
"push   r11                 \n\t"   \
"push   r12                 \n\t"   \
"push   r13                 \n\t"   \
"push   r14                 \n\t"   \
"push   r15                 \n\t"   \
"push   r16                 \n\t"   \
"push   r17                 \n\t"   \
"push   r18                 \n\t"   \
"push   r19                 \n\t"   \
"push   r20                 \n\t"   \
"push   r21                 \n\t"   \
"push   r22                 \n\t"   \
"push   r23                 \n\t"   \
"push   r24                 \n\t"   \
"push   r25                 \n\t"   \
"push   r26                 \n\t"   \
```

```
"push   r27                     \n\t"   \
"push   r28                     \n\t"   \
"push   r29                     \n\t"   \
"push   r30                     \n\t"   \
"push   r31                     \n\t" /* Save r31 content on the stack */  \
"lds    r26, context            \n\t" /* Put *context in r26 (low byte of X) */ \
"lds    r27, context + 1        \n\t" /* Put *(context+1) in r27 (hight byte of X) */
"in     r24, __SP_L__           \n\t" /* Put SPL content in r24 */  \
"in     r25, __SP_H__           \n\t" /* Put SPH content in r25 */  \
"adiw   r24, 0x1                \n\t" /* Add 1 to r24 (Saved SPL */  \
"st     x+, r24                 \n\t" /* Store r24 content at X */  \
"st     x+, r25                 \n\t" /* Store r25 content at X+1 */  \
);
```

The `SAVE_CONTEXT()` basicaly add on the stack `r0`, `SREG`, registers from `r1` to `r31`, and then put `SPL` and `SPH` at the adress pointed by `context`. (table 5)
From here the current context is saved on the current stack, and the `old_context` pointer given as argument, points to the top of the current stack.

After testing the validity of `new_context`, showing some debugging symbols, we set the global variable `context` to `new_context`, and call the `RESTORE_CONTEXT()` macro. Obviously `RESTORE_CONTEXT()` mission is the opposite of the `SAVE_CONTEXT()` one. Assuming that the adress pointed by `new_context` pointer given as argument points to the top of `thread1` stack, the exact opposite steps of `SAVE_CONTEXT()` will be reproduced.

```
#define RESTORE_CONTEXT()  \
asm volatile ( \
"lds    r26, context            \n\t"  /* Load *context in r26 (XL)*/\
"lds    r27, context + 1        \n\t"  /* Load *(context+1) in r27 (XH)*/ \
"ld     r28, x+                 \n\t"  /* Load *X in r28 and X=X+1 */\
"ld     r29, x+                 \n\t"  /* Load *X in r29 */\
"sbiw   r28, 0x1                \n\t"  /* Remove 1 to r28 */ \
"out    __SP_L__, r28           \n\t"  /* Put r28 content in SPL */  \
"out    __SP_H__, r29           \n\t"  /* Put r29 content in SPH */  \
"pop    r31                     \n\t"  /* Put top of the stack in r31 */ \
"pop    r30                     \n\t"   \
"pop    r29                     \n\t"   \
"pop    r28                     \n\t"   \
"pop    r27                     \n\t"   \
"pop    r26                     \n\t"   \
"pop    r25                     \n\t"   \
"pop    r24                     \n\t"   \
"pop    r23                     \n\t"   \
```

... ... current thread0 stack

Table 1: Before jumping in `switchContext` .

... ... current thread0 stack
PC3 PC2 PC1

Table 2: Just after jumping in `switchContext` .

... ... current thread0 stack
PC3 PC2 PC1
PC3bis PC2bis PC1bis ... ... more printf stuffs ... ...

Table 3: While executing `printf` .

... ... current thread0 stack
PC3 PC2 PC1

Table 4: Juste after returning from `printf` .

... ... current thread0 stack
PC3 PC2 PC1
r0 SREG r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 r16 r17 r18 r19 r20 r21 r22 r23 r24 r25 r26 r27 r28 r29 r30 r31

Table 5: Juste after executing `SAVE_CONTEXT()` . .

```
"pop    r22                     \n\t"   \
"pop    r21                     \n\t"   \
"pop    r20                     \n\t"   \
"pop    r19                     \n\t"   \
"pop    r18                     \n\t"   \
"pop    r17                     \n\t"   \
"pop    r16                     \n\t"   \
"pop    r15                     \n\t"   \
"pop    r14                     \n\t"   \
"pop    r13                     \n\t"   \
"pop    r12                     \n\t"   \
"pop    r11                     \n\t"   \
"pop    r10                     \n\t"   \
"pop    r9                      \n\t"   \
"pop    r8                      \n\t"   \
"pop    r7                      \n\t"   \
"pop    r6                      \n\t"   \
"pop    r5                      \n\t"   \
"pop    r4                      \n\t"   \
"pop    r3                      \n\t"   \
"pop    r2                      \n\t"   \
"pop    r1                      \n\t"   \
"pop    r0                      \n\t"  /* Restore r0 from the stack*/ \
"out    __SREG__, r0       \n\t"  /* Restore SREG from r0 */ \
"pop    r0                      \n\t"  /* Restore the real r0 */ \
);
```

After restoring SP, the general purpose registers, SREG, we use the ret instruction to restore PC form the stack. The execution of thread1 can then go on from where it was suspended before.

2.2.2 Initialising a new thread

We will now look at the procedure we use to initialise a new thread from scratch. What we need is a working context that can be restored using the RESTORE_CONTEXT() macro. Obviously when starting a thread from scratch most of the registers will contain dummy values that won't get used. To ease debugging, we choosed to fill them with recognisable values : r11 will contain 11, r3 3 and so on. Thus some registers need some special attention :

- r1 needs to be 0 while executing C code
- r25 to r8 are used to pass function arguments. Here we planed to allow to start the thread by a function using one argument of type any (a pointer). Which would be passed on r25 and r24.

Remembering that the stack starts at the top of it's allocated space and grows downward, one can understand the following code :

```
void SRV_trap__initContext(jbyte* virtcontext,
                           any code_entry,
                           jbyte stackbase[],
                           jint stacksize,
                           any code_entry_argument,
                           jboolean user) {
    jushort temp_address;

    DEBUG_PRINTF("SRV_trap__initContext \n");

    // Let's start to write at the begining of the stack
    // which is the end of the memory region.

    temp_address = (jushort) code_entry;
    // Write PC-1 on the stack
    stackbase[--stacksize] = (jbyte) ( temp_address & (jushort) 0x00ff );
    // Write PC-2 on the stack
    temp_address >>= 8;
    stackbase[--stacksize] = (jbyte) ( temp_address & (jushort) 0x00ff );
    // Write PC-3 on the stack. Remember PC is 24 bits on atm2561
    stackbase[--stacksize] = 0;

#warning I can't find out why code_entry wouldn't be more than exp(2,16).
// Seems to work anyway, but wasn't tested a lot.

    stackbase[--stacksize] = (jbyte) (  0 ); /* R0 */
    // Start tasks with interrupts enabled.
    stackbase[--stacksize] = (jbyte) ( 0x80 ); /* SREG */
    // The compiler expects R1 to be 0.
    stackbase[--stacksize] = (jbyte) ( 0x0 ); /* R1 */
    stackbase[--stacksize] = (jbyte) (  2 ); /* R2 */
    stackbase[--stacksize] = (jbyte) (  3 ); /* R3 */
    stackbase[--stacksize] = (jbyte) (  4 ); /* R4 */
    stackbase[--stacksize] = (jbyte) (  5 ); /* R5 */
    stackbase[--stacksize] = (jbyte) (  6 ); /* R6 */
    stackbase[--stacksize] = (jbyte) (  7 ); /* R7 */
    stackbase[--stacksize] = (jbyte) (  8 ); /* R8 */
    stackbase[--stacksize] = (jbyte) (  9 ); /* R9 */
    stackbase[--stacksize] = (jbyte) ( 10 ); /* R10 */
    stackbase[--stacksize] = (jbyte) ( 11 ); /* R11 */
```

```
    stackbase[--stacksize] = (jbyte) ( 12 ); /* R12 */
    stackbase[--stacksize] = (jbyte) ( 13 ); /* R13 */
    stackbase[--stacksize] = (jbyte) ( 14 ); /* R14 */
    stackbase[--stacksize] = (jbyte) ( 15 ); /* R15 */
    stackbase[--stacksize] = (jbyte) ( 16 ); /* R16 */
    stackbase[--stacksize] = (jbyte) ( 17 ); /* R17 */
    stackbase[--stacksize] = (jbyte) ( 18 ); /* R18 */
    stackbase[--stacksize] = (jbyte) ( 19 ); /* R19 */
    stackbase[--stacksize] = (jbyte) ( 20 ); /* R20 */
    stackbase[--stacksize] = (jbyte) ( 21 ); /* R21 */
    stackbase[--stacksize] = (jbyte) ( 22 ); /* R22 */
    stackbase[--stacksize] = (jbyte) ( 23 ); /* R23 */

    /* Place the parameter on the stack in the expected location. */
    temp_address = (jushort) code_entry_argument;
    stackbase[--stacksize] = (jbyte) ( temp_address & (jushort) 0x00ff );
    temp_address >>= 8;
    stackbase[--stacksize] = (jbyte) ( temp_address & (jushort) 0x00ff );

    stackbase[--stacksize] = (jbyte) 26;  /* R26 X */
    stackbase[--stacksize] = (jbyte) 27;  /* R27 */
    stackbase[--stacksize] = (jbyte) 28;  /* R28 Y */
    stackbase[--stacksize] = (jbyte) 29;  /* R29 */
    stackbase[--stacksize] = (jbyte) 30;  /* R30 Z */
    stackbase[--stacksize] = (jbyte) 31; /* R31 */

    temp_address = (jushort) stackbase + stacksize;
    *virtcontext = (jbyte) ( temp_address & (jushort) 0x00ff );
    temp_address >>= 8;
    *(virtcontext+1) = (jbyte) ( temp_address & (jushort) 0x00ff );
}
```

Where :

- `virtcontext` is the address of a pointer which points to saved SP values.
- `code_entry` is a pointer to the function which should start the thread.
- `stack_base` is the begining of the memory region which will contain the stack.
- `stacksize` is the allocated size of the stack. We use 256 on the atm2561.
- `code_entry_argument` is passed to `code_entry` function pointer as argument. We don't use it yet.

- `user` isn't used yet.

Using `RESTORE_CONTEXT()` on a thread initialised with the `init_context` method, will then start executing `code_entry`.

3 Higher level facilities

3.1 v2 style Job management

The folowing is related to the implementation of the Job component and it's related components : Scheduler, Tick, Semaphores, Timeout. All of those have been ported in a straight an forward way from v2. One particularity of this implementation is that each job keeps a reference of the previews and next job. Thus creating an oriented chain of job. This way no component needs to hold a table with the jobs he needs to refer to. Jobs can be easily re-ordered, inserted or removed from job chaines. The drawback is that each job manipulation requires many call on Job interfaces, dynamic binding ...

3.1.1 The Job component

The role of the `job` component is to store various information regarding a thread. As private datas we have :

```
struct {
  // handle to previous and next job
  any prev, next;
  // the stack that will be used by the job runner
  jbyte stack[256];
  // place to save this stack pointer
  jbyte context[2];
} PRIVATE;
```

And here comes the `job` component ADL :

```
component avr.activity.lib.job {
  provides activity.api.Job as job
  provides fractal.api.LifeCycleController as lcc

  requires activity.api.Scheduler as scheduler
  // Bind the functional code of your thread to
  // this runner.
  requires activity.api.Runner as runner //optional
    attribute  jint autorun = 0
    attribute  jint runasuser = 0
```

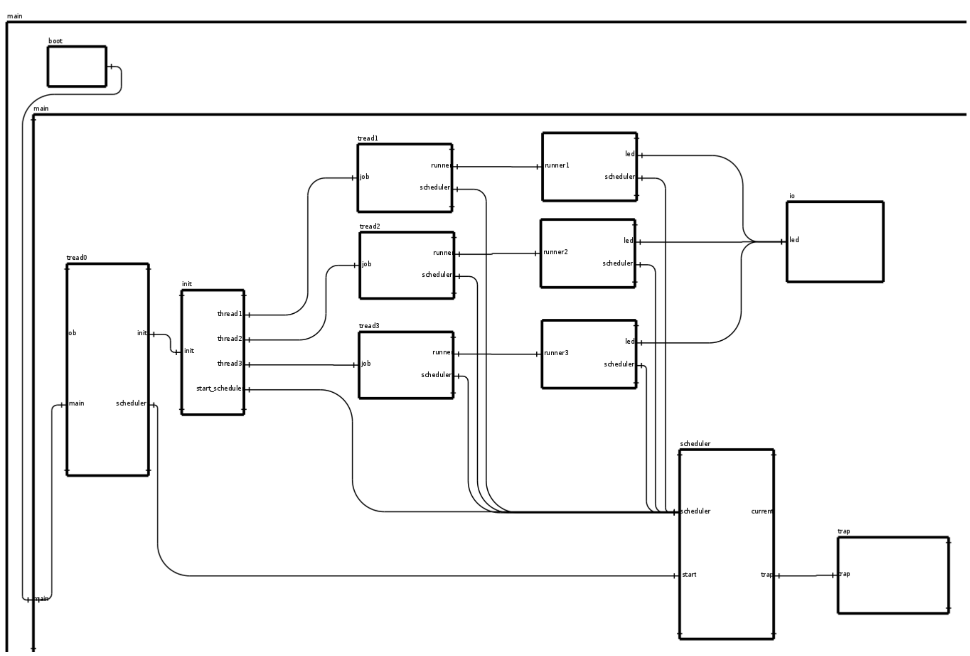
```
    attribute  jint priority = 0
    attribute  jint generation = 0

  content avr.activity.lib.job
  // [shared = false] tag is needed to prevent
  // the compiler to had the "compId" argument
  // to private methods.
  implementation default [shared = false]
}
```

Attributes `autorun`, `runasuser` and `generation` are unused yet on AVR. Attribute `priority` is exposed throught a server interface method, and is used by component such as a scheduler.

3.1.2 A cooperative scheduling example

We're gonna examine how to compose with the `trap`, `job` and `scheduler` components to create a demo application. In this application, we will create three threads with the same priority, which do light some leds, busy lock and then ask for a re-scheduling. This functional code is encapsulated behind a `Runner` interface, which is intended to be bound to a `job` component.
The architecture we set is described on figure 1.



,

Figure 1: Representation of the threads_cooperative example ADL.

To summarize we have three `job` components. Each of those is bound to its own functional code, which will be run in it's own context, using it's own stack. At that point one might notice that the initial stack is not used anymore. Anyway we will use a fourth job component to retain information about the intial stack. Howether this component is a bit different of the `job` component.

- It doesn't need to be initialised as everything that happens from the boot to the first context change was done in it's own context.
- It should not be destroyed.
- It should never return.

For this we use the component `initjob`, which also provides the main entry. It's functional code will have as a dutty to initialize the other jobs, and start the scheduler. Let's now try to follow the execution flow for all thoses threads.

Initial thread execution flow : After booting and executing start methods of all lifecycle controllers, it jumps in `initjob` main entry.

```
void SRV_main__main(jint argc, jstring *argv) {
    // Register ourself on the scheduler.
    CLT_scheduler__addJob(SRVID_job);
    // Execute our functional code.
    CLT_init__run();
    // Hold the execution in case everything returns
    while (1) {
        DEBUG_PRINTF("Looping in initjob !\n");
    }
}
```

As it is the first job to be created, registering it on the scheduler is trivial. We'll look at the scheduler latter.
It'll then jump into it's functional code. Which will start the other jobs.

```
void SRV_init__run( ) {
    // Start all child tasks
    CLT_thread1__run();
    CLT_thread2__run();
    CLT_thread3__run();
    // Start the scheduler
    CLT_start_scheduler__run();
    while(1) {
        DEBUG_PRINTF("Executing init Job ! \n");
    }
};
```

Which asks to get initialised by the scheduler.

```
void PRV_run() {
  // Ask the scheduler to initalize this job and its stack
  CLT_scheduler__initJob( SRVID_job,
                          &PRV_dummy_start,
                          &PRIVATE.stack,
                          256,
                          0x0,
                          ATT_runasuser);
  return;
}

void SRV_job__run() {
  PRV_run();
  return;
}
```

Which in turns ask to the `trap` component to initialize a stack as described in 2.2.2 and register the job.

```
void SRV_scheduler__initJob( any job_itf,
                             any function,
                             any stackbase,
                             jint stacksize,
                             any arg,
                             jboolean user) {
    jbyte* convenience_context;
    CLT_bc__bind("convenience_job", job_itf);
    convenience_context = CLT_convenience_job__getContext();
    DEBUG_PRINTF("SRV_scheduler__initJob, job %x, context %x \n", job_itf, convenience
    CLT_trap__initContext( convenience_context, function, stackbase, stacksize, &arg,
    PRV_addJob( job_itf );
}

void PRV_addJob(any job_itf) {
    jint priority;
    any next_job_itf;
    any prev_job_itf;

    DEBUG_PRINTF("PRV_addJob %x ", job_itf);

    CLT_bc__bind("convenience_job",job_itf);
```

```
    priority = CLT_convenience_job__getPriority();
    DEBUG_PRINTF("with priority %d \n", priority);

    if(PRIVATE.runnable_job[priority] == 0) {
        // I'm alone in the queue
        CLT_convenience_job__setNext( job_itf );
        CLT_convenience_job__setPrev( job_itf );
        // then i'm the head
        PRIVATE.runnable_job[priority] = job_itf;
    } else {
        // Insert me at the last place
        next_job_itf = PRIVATE.runnable_job[priority];

        CLT_bc__bind("convenience_job", next_job_itf);
        prev_job_itf = CLT_convenience_job__getPrev();
        CLT_convenience_job__setPrev( job_itf );
        CLT_bc__bind("convenience_job", prev_job_itf );
        CLT_convenience_job__setNext( job_itf );


        // Link myself to neighbour
        CLT_bc__bind("convenience_job", job_itf);
        CLT_convenience_job__setPrev( prev_job_itf );
        CLT_convenience_job__setNext( next_job_itf );

        // Put me at the first place
        PRIVATE.runnable_job[priority] = job_itf;
    }
    return;
}
```

After initialising all jobs this way, the scheduler get started.

```
void SRV_start__run() {
    DEBUG_PRINTF("\n********** Starting scheduler ! **********\n \n");
    PRIVATE.started = 1;
    PRV_yield();
}
```

A new job is chosed, and contexts get changed.

```
void PRV_yield() {
    jbyte* current_context;
    jbyte* new_context;
```

```
    DEBUG_PRINTF("PRV_yield \n");

      DEBUG_PRINTF("Current job : %x \n", CLT_bc__lookup("current_job"));
      current_context = CLT_current_job__getContext();
      new_context = PRV_electone();
      CLT_trap__switchContext( current_context, new_context );
}

void SRV_scheduler__yield() {
    PRV_yield();
}
```

As we took care to set the `initjob` priority to zero and other the `job` priority to a higher value, `initjob` sould never be elected again.

A normal thread execution flow : So a `job` has been given some processing time. `switchContext` (2.2.1) has been called.
The first time it gets called, the initial context will be restored. As `PC` has been set to point to `PRV_dummy__start()`, we'll jump there. (Remember we wrote it at the very begining of the stack (2.2.2))

```
void PRV_dummy_start() {
  // Let's call the thread functional code
  CLT_runner__run();

  // We returned from functional code.
  // The thread did his dutty.
  // Let' destroy it now
  CLT_scheduler__destroyJob();

  DEBUG_PRINTF("Must never pass here !\n");
  return;
}
```

And then the functional code get called.

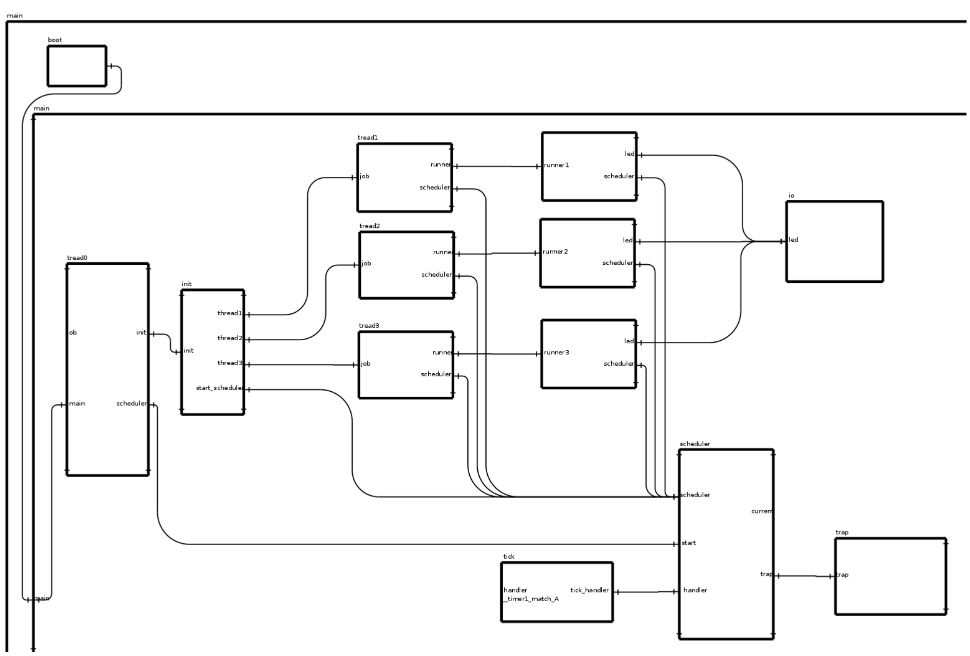
```
void SRV_runner__run() {
juint i;
    while(1) {
        // Light some leds to give a visual
        // notification of which thread is executing
        CLT_led__setLEDs( ATT_leds_to_light_up );
        // Busy wait
        for (i=0; i<60000; i++) {
```

```
            __asm__ __volatile__ (  "nop" );
        }
        // Give some execution time to other threads
        CLT_scheduler__yield();
    }
}
```

After lighting up some leds, `CLT_scheduler__yield` is called. A new context is chosen, and then the new context and the current one are switched as explained in 2.2.1.
Next time this context will be set, execution will restart from the end of `SRV_trap__switchContext`, and return through `PRV_yield`, `SRV_scheduler__yield`, and loop one more time in the `while(1)` ...

3.1.3 A preemptive scheduling example

We will now describe an example using the almost same functional code, but using preemptive scheduling. Cooperative capabilities as described in 1 are still usable, but we chose here to rely on an external tick to decide when to change the active thread. The tick is given by the `tick` component, which uses the physical timer1 on the AVR. The architecture we set is described on figure 2.



,

Figure 2: Representation of the threads_preemptive example ADL.

The `tick` component will emit a `__timer1_match_A` interupt at a given frequency. This frequency is given as an attribute `frequence` in the ADL.

Assuming every thing as been initialised as in the cooperative example, we can follow the execution flow of a thread. We'll start from the occurence of a tick : from the `__timer1_match_A` interupt handler.

```
/*
 * @@ServerMethod(tickhandler, execute) @@
 * @@ KeepName @@
 */
__attribute__ ((signal, naked)) void __timer1_match_A();
void __timer1_match_A() {
    SAVE_CONTEXT_FROM_TICK();
    // Now that we saved the stack, everything we'll do from here
    // won't harm. We should never return from this function.

    // Set next tick
    _SFR_WORD(OCR1A) = (volatile) _SFR_WORD(OCR1A) + PRIVATE.tick;
    // Call the scheduler
    DEBUG_PRINTF("\n__timer1_match_A \n");
    DEBUG_PRINTF("Saved current context at %x \n", context);
    CLT_timerhandler__execute(); // This should at least be quicker than PRIVATE.tick


    // Actually as this interrupt
    // is used as a tick,
    // we should never return from
    // CLT_timerhandler__execute()
    RESTORE_CONTEXT();
    __asm__ __volatile__ ("ret");
}
```

We use here two attributes for `__timer1_match_A` :

- `signal` which specify that this function is an interupt handler.
- `naked` which tells GCC that it shouldn't modifie the stack while entering this function. (Obviously `PC` is written on the stack anyway, but nothing else is.)

First thing we do when entering the tickhandler, is to save the context on the stack. We don't use `SAVE_CONTEXT()` here but `SAVE_CONTEXT_FROM_TICK()`. Both are almost identical. We will restore both using `RESTORE_CONTEXT()`. The difference is that with `SAVE_CONTEXT_FROM_TICK()` we do set the interuption flag as it was before the execution got interrupted : enabled. That way we will restore the exact same context latter. The global variable `context`, is allready pointing to the current job context pointer. We don't need to set it again.

We can then set the timer for the next tick, and call the timer handler which is the scheduler.

```
void SRV_handler__execute() {
    jbyte* new_context;

    // Here, we should be
    // on an interrupt context.
    DEBUG_PRINTF("SRV_handler__execute \n");
    if (PRIVATE.started == 0 ) {
      DEBUG_PRINTF("Received a tick but scheduler isn't started yet ! \n");
    }
    // Select the next thread to schedule
    new_context = PRV_electone();
    // and move to its execution context
    CLT_trap__setContext( new_context );

    // Actually we should not return
    // from CLT_trap__setContext()
    return;
}
```

The scheduler then elect the next job which deserves cpu time, and ask to the `trap` component to set it.

```
__attribute__ ((naked)) void SRV_trap__setContext(jbyte* new_context) {
    jubyte i;

    DEBUG_PRINTF("SRV_trap__setContext \n");

    // Check if interuptse are enbled or not
    if( PRV_InInterrupt() == 1 ) {
        // Make the global variable context
        // point on the new stack pointer
        // to use it the assembly
        // RESTORE_CONTEXT macro.
        context = new_context;
        // set the new context
        RESTORE_CONTEXT();
        // naked function. Need manual ret
        __asm__ __volatile__ ("ret");
    }
    else {
```

```
        // If not in interrupt context, who did save current context ???
        // Maybe we should set the new context anyway instead of returning ???
        // Feel free to change this behavior !
        DEBUG_PRINTF("Using setContext outside an interrupt context ??? \n");
        __asm__ __volatile__ ("ret");
    }
}
```

Note that we do lost everything that happened from the tick to `RESTORE_CONTEXT()`. We won't return from `SRV_trap__setContext` nor `SRV_handler__execute` (from the scheduler).