Using Nuptse on AVR

Julien TOUS

October 1, 2007

1 introduction

This document aims at presenting the creation of simple (ie stupid) kernels for the AVR Atmega2561 using the Nuptse version of THINK. To get information about the avr-toolchain and the Nuptse compiler, you might want to read the "Think-nuptse-doc" `https://yourdev.rd.francetelecom.fr/svn/viewvc.php/papers/doc/`, the "Introduction to the AVR/Cognichip platform" on Codex `http://yourdev.rd.francetelecom.fr/docman/` and the Atmega2561 Datasheet.
To build the examples, you should get a copy of Nuptse and Kortex on ObjectWeb svn :

```
 svn checkout svn://svn.forge.objectweb.org/svnroot/think/nuptse/trunk  svn
checkout svn://svn.forge.objectweb.org/svnroot/think/kortex/branches/avr
```

And define `$KORTEX_PATH` and `$THINK_PATH` according to the place you "checkout-ed" each project.

2 HelloWorld and debuging

2.1 Compiling and running

Get to the example directorie :

```
# cd $KORTEX_PATH/../examples/avr/atm2561/helloworld
```

Compile example :

```
# ant avr
```

Open a serial terminal :

```
# gtkterm
```

Load the compiled kernel in a debuger :

```
# ice-gdb kernel
```

And run it :

```
(gdb) c
```

If everything is configured correctly, you can see in the serial terminal :

```
HelloWorld from Nuptse!
```

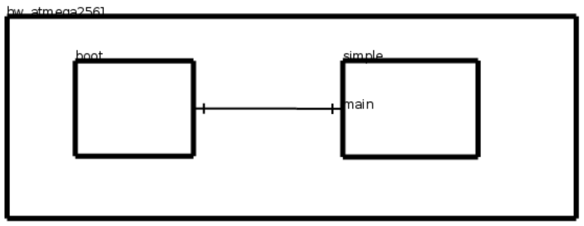
2.2 What does really happen

When you compile `# ant avr` ant looks for a section starting with `<target name="avr" ... />` in a file named `build.xml` :

```
<project name="Helloworld example compilation" default="avr">

<property environment="env" />
<property file="../../../common/compiler.properties" />
<property file="../common/compiler.properties" />
<property file="${basedir}/build.properties" />

  <target name="avr" description="Compile Helloworld example for the ATmega2561">
    <mkdir dir="${out-path}" />
    <ant antfile="${thinkadl.root}/execute.xml" target="compile">
      <property name="target" value="avr.hw_atmega2561" />
      <property name="properties_path"
                value="../common/common.properties:avr.properties:build.properties" />
      <property name="verbose.level" value="INFO" />
    </ant>
      <exec executable="avr-objcopy">
        <arg line="-O binary build/obj_atm2561/hw_atmega2561 kernel.bin" />
      </exec>
    <exec executable="cp">
       <arg line="build/obj_atm2561/hw_atmega2561 kernel" />
     </exec>
  </target>

  <target name="clean" description="remove generated files">
    <delete dir="${out-path}" />
    <delete>
      <fileset dir="." includes="core*" />
    </delete>
  </target>

</project>
```

The line `<property name="target" value="avr.hw_atmega2561" />` tells the compiler to compile file situated at `./src/avr/hw_atmega2561`.

Let's have a look at the avr kernel code in `./src/avr`. The architecture of the kernel can be visualised on the figure 1 :
Component **hw_atmega2561** contains two subcomponents : **boot** and **simple**. This is described in `./src/avr/hw_atmega2561.adl` :



,

Figure 1: Representation of the helloworld example ADL.

```
component hw_atmega2561 {
   contains simple = avr.simple
   contains boot = avr.atm2561.boot.lib.boot
   binds boot.entry to simple.main
}
```

Component **boot** belongs to the kortex library which provides a lot of ready to use components. Component **simple** as been written specifically for this example :

```
 component simple {
    provides activity.api.Main as main
    content avr.simple
}
```

Line two specifies that the component has a server interface named main which implement IDL at `$KORTEX_PATH/generic/activity/api/Main.idl` Line three specifies that the c code implementing this ADL is in `./src/avr/simple.c` :

```
#include "debug.h"

void SRV_main__main(jint argc, jstring* argv) {
    DEBUG_PRINTF("HelloWorld from Nuptse!");
}
```

After booting, component **boot** runs the **main** method on the **main** interface, which is provided by simple and simply does : `DEBUG_PRINTF("HelloWorld from Nuptse!")`

`DEBUG_PRINTF` is a macro which is defined in `$KORTEX_PATH/arch/avr/libc/debug.h`. By default it does nothing. But if you define `PRINTF` it is replaced with `printf`. In this example `DEBUG_PRINTF` has been activated in file `./build.properties` with the line : `defs -DPRINTF`.

Important note : In order to use printf on a serial terminal, avr-libc has been slightly modified (run `# grep -R mystdout $KORTEX_PATH/src/avr/libc` to see modified files). You should keep in mind that this is not a proper way to send chars on the serial port. This should just be used for debuging purpose. It just allows us to use `printf` for debuging purpose and remove it from final kernels without modifying the architecture.

NB : Using printf in a kernel also includes a lot of stuff from avr-libc and so increase the kernel size and compilation time quite a lot.

3 Helloleds

We're now gonna look at another example which will use the most visible feature of the Cognichip : the leds. `# cd $KORTEX_PATH/../examples/helloleds`

3.1 The Led interface and io component

The architecture of this kernel can be visualised on the figure 2 :

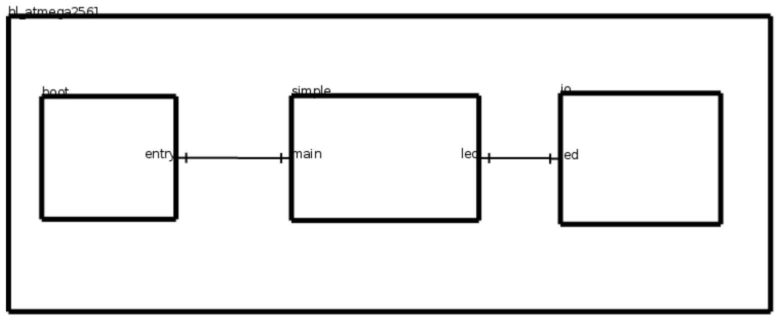


Figure 2: Representation of the helloleds example ADL.

We added a component **io** bound to the component **simple** throught an interface **Led**. The interface **Led**, is described in by the IDL at `./src/avr/api/Led.idl` :

```
package avr.hw.api;

public interface Led {
   void setLEDs(unsigned byte state);
   unsigned byte getLEDs();
}
```

We can see that this interface provides two methods : `setLEDs` and `getLEDs`. These methods are implemented in the C file `$KORTEX_PATH/arc/avr/hw/lib/io.c`.

```
#include <avr/api/io.h>
#include "debug.h"

void SRV_led__setLEDs( jubyte state) {
    PORTC=state;
    DEBUG_PRINTF("Leds have been actualised \n");
}

jubyte SRV_led__getLEDs() {
    return PORTC;
}
```

`state` argument is of type `jubyte` which is 8 bit long. `PORTC` points on the adress of the leds on the AVR. Each bit of `PORTC`, represent a led. The led is on if the corresponding bit is set to `0`, off if set to `1`.
On the client side, those method are called from `./src/avr/simplet.c` to light on all the leds.

```
#include "debug.h"

void SRV_main__main(jint argc, jstring* argv) {
    DEBUG_PRINTF("Entered simple.c \n");
    CLT_led__setLEDs(0);
}
```

4 moving_leds

In this example we will create some visual effects with the leds : Let's light up all the leds but one, and change periodicaly the dead led so it looks like it's moving to the right.
Let's first have a glance at the architecture of this example on figure .
The "execution flow" is as follow :

- **boot** component calls the main method on the composite component **movingleds_atmega2561** which is forwarded to the **T** component.

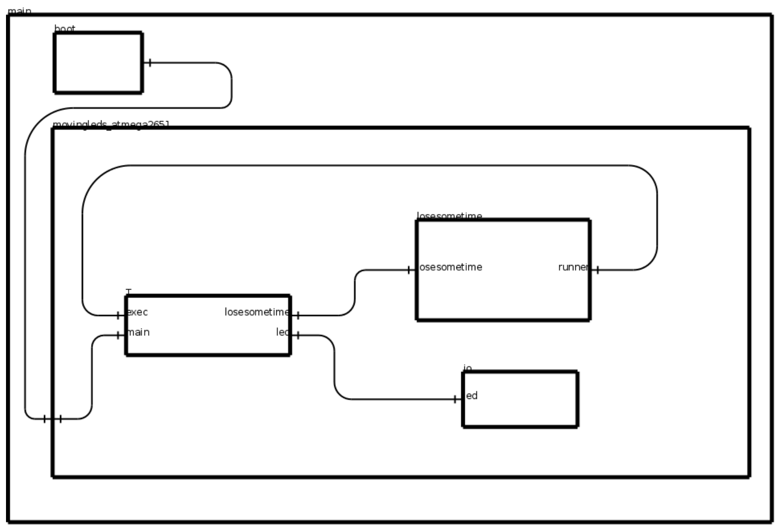


Figure 3: Representation of the movingleds example ADL.

- **T** component calls the **setLEDs** method on interface **led** to light all the leds but the first. When the leds are on, it then calls **losesometime** component thrue the **losesometime** interface.
- Component **losesometime** wastes some time by doing a lot of unusefull operations and then calls component **T**.
- **T** component changes the position of the dead led and, calls the **setLEDs** method on interface **led** to render it, and then call component **losesometime** again.
- ...
- It goes like this until the pile gets full.

NB : The role of the **losesometime** component is to get a visual effect. Without a latency between two calls on component **T** leds would be changed so often that we would see as if they were all lighted.

4.1 The Handler interface

As we can see on `./src/avr/losesometime.adl`, the **losesometime** component provides the **irq.api.alarm** IDL for inteface **losesometime** and requires also **generic.irq.handler** for the **handler** interface.
The **Handler** IDL is given bellow :

```
package irq.api;

public interface Handler {
   void execute();
}
```

It only implements one method (**execute**) which takes no arguments.

4.2 Size of a jubyte

Let's now have a look at the **T** component :

```
struct {
volatile jubyte counter;
} PRIVATE;

void PRV_light_the_leds() {
PRIVATE.counter = PRIVATE.counter * 2;
    if (PRIVATE.counter == 0) {
        PRIVATE.counter =1;
    }
CLT_led__setLEDs(PRIVATE.counter);
CLT_losesometime__set(10000);
}

/*
 * exec interface methods
 */

void SRV_handler__execute() {
  PRV_light_the_leds();
}

/*
 * main interface methods
 */
void SRV_main__main(jint argc, jstring* argv) {
PRV_light_the_leds();
}
```

The main part of the code is located in the function `light_the_leds()` which is called both at the start (when the **main** method is called) and at every iteration.
To understand the way `light_the_leds()` works, you must keep in mind that `PRIVATE.counter` is a `jubyte` and so can represent a value from 0 up to 255.

- The first time `light_the_leds()` is called `PRIVATE.counter` worths `0` and so is set to `1` by the `if` statement : binary value of `PRIVATE.counter` is `00000001`. The first led is off, all the others are on.

- Next time we call `light_the_leds()`, `PRIVATE.counter` becomes `2` which in binary is `00000010`. The second led is off, all the others are on.

- At the height iteration, `PRIVATE.counter` is `128` (`10000000`), and should become `256` (`1 00000000`) which needs 9 bits to get represented and so get troncated and really becomes `00000000` and then after the `if` statement it becomes `1`.

- ...

5 timerenabled_movingleds

As said previously component **losesometime** adds a latency between two calls on component **T** by wasting cpu time. It's a busy lock. Althought this is a very easy way to get a visual effect, it is neither precise nor elegant. We are now going to replace that ugly busylock by an alarm that will raise an interuption at a given time.
The architecture of this example represented on figure 4 hasn't changed a lot. **losesometime** component as been replaced by the couple of component **trap** and **timer**. **timer**'s ADL is shown bellow.

```
component avr.atm2561.irq.lib.timer0 extends avr.atm2561.irq.lib.timerType {
  content avr.atm2561.irq.lib.timer0
}
```

We can see on line one, with the keyword `extends` that this ADL is just an implementation of a more generic component **timerType**.

```
abstract component avr.atm2561.irq.lib.timerType  {
  requires irq.api.Handler as timerHandler optional
  provides irq.api.Handler as handler
  provides irq.api.Alarm as timerControl
  attribute julong cpu_freq
}
```

The **Alarm** interface has three methods : **set** which take as argument the time in millisecond we should wait before receiving an interuption. an **unset** which takes no argument and simply removes a remaining alarm. And **remain** which returns the time remaining until incoming interuption.

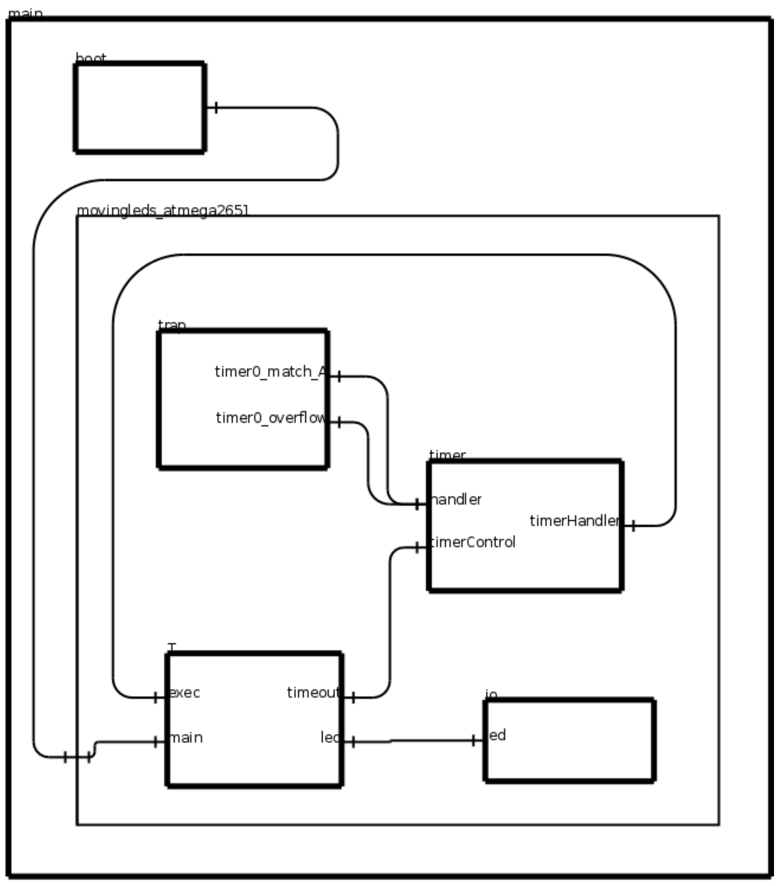


Figure 4: Representation of the timerenabled_movingleds example ADL.

```
package irq.api;

public interface Alarm {
   void set(unsigned int millisecond);
   void unset();
   unsigned int remain();
}
```

5.1 The trap component

The role of the **trap** component is to catch interuption. When an interuption get catched, **trap** disable interuptions, saves the state of the registers, and call the appropriate handler of interuption. When the handler finishes its work, **trap** restore previously stored registers, re-enables interuption, and the execution can go on from

where it was stoped.
On our example we only watch two interuptions : `timer0_match_A` and `timer0_overflow`.
But the Atmega2561 has 56 different interuptions.

5.2 The timer component

The role of the **timer** component on this example is to configure the internal timer0 of the Atmega2561 to send an interuption that will be the start signal for **T** component to update the leds status.
One of the shortcomming of the internal timer0 counter `TCNT0` is that it is coded on a byte and so can only count till 255. It however can use a prescalar which can be set to `1`, `8`, `64`, `256`, or `1024`, meanning that it can count respectively every cpu-tick, every second cpu-tick, every 8-th cpu-tick, ... or 1024-th cpu-tick.
Now that we have the esential informations we can have a look at the **timer** code

The **set** method of the **Alarm** interface, does nothing but convert argument given in millisecond to a number of cpu-tick and call function `reset()`.

```
void SRV_timerControl__set(juint millisecond) {
  // convert to cpu ticks;
  PRIVATE.rem_cputicks = ((julong) (ATT_cpu_freq / 1000) ) * millisecond;
  reset();
}
```

Function `reset()` really set an interuption. We will make a distinction between two cases :

1. If the number of cpu-tick to wait is lower than `255`, `reset()` sets the prescalar to 1, enable interuption on compare (`timer0_match_A`) and set the compare value.
2. If the number of cpu-tick to wait is bigger than `255`, `reset()` enable interuption on overflow (`timer0_overflow`) and takes care of choosing the biggest prescalar possible in order to get an overflow interuption before the alarm.

```
void reset() {
  julong rem;
  jbyte tcnt = 0;
  rem = PRIVATE.rem_cputicks;
  // set prescalar to 0
  TCCR0B = 0;
  // choose prescalar mask
  if (rem >= (julong) 1024 * 256) {
   tcnt = 0x05;
  }
    else if (rem >= (julong) 256 * 256) {
    tcnt = 0x04;
  }
    else if (rem >= 64 * 256) {
    tcnt = 0x03;
  }
    else if (rem >= 8 * 256) {
    tcnt = 0x02;
  }
    else if (rem >= 256) {
    tcnt = 0x1;
  }
  else {
    // set compare register
    OCR0A = rem;
    // clear counter
    TCNT0 = 0;
    // set prescalar to 1
    tcnt = 0x01;
    TCCR0B = tcnt;
    // disable interrupt on overflow, enable interrupt on compare
    SET(TIMSK0, OCIE0A);
    return 0;
  }
  // clear counter
  TCNT0 = 0;
  // set prescalar;
  TCCR0B = tcnt;
  // enable interrupt on overflow, disable interrupt on compare
  SET(TIMSK0, TOIE0);
  return 0;
}
```

When one or the other of those interuption get catched by the **trap** component, the **execute** method from the **Handler** interface is called.

```
void SRV_handler__execute() {
  // Save prescalar value
  jubyte prescalar = TCCR0B & 0x07;
  // Stop timer and set prescalar to 0
  TCCR0B = 0;
  // Disable timer0 iteruptions
  UNSET(TIMSK0, OCIE0A);
  UNSET(TIMSK0, TOIE0);
  //clear counter
  TCNT0 = 0;
  // Update number of cpu tick remaining until alarm time
  // according to prvious prescalar value.
  if (prescalar == 0x05) {
    PRIVATE.rem_cputicks -= (julong) 256 * 1024;
  }
  else if (prescalar == 0x04) {
    PRIVATE.rem_cputicks -= (julong)  256 * 256;
  }
  else if (prescalar == 0x03) {
    PRIVATE.rem_cputicks -= (julong)  256 * 64;
  }
  else if (prescalar == 0x02) {
    PRIVATE.rem_cputicks -=  (julong) 256 * 8;
  }
  else if (prescalar == 0x01) {
    // Prescalar was previously set to 1
    if (PRIVATE.rem_cputicks > 256) {
      // Their is still some time to go until alarme time.
      // The interuption we cought must be an overflow
      PRIVATE.rem_cputicks -= 256;
    }
    else {
      // The interuption we cought must be match_A
      // We can call the handler
      CLT_timerHandler__execute();
      return 0;
    }
  }
  reset();
}
```

1. If it received a `timer0_match_A` interuption. This is the interuption that component **T** asked for. The client handler is called. Component **T** will ”wake up”.

2. If it received a `timer0_overflow` interuption. Depending on the actual prescalar, the number of cpu-tick to remain is updated and a new interuption is set by

calling the `reset()` function.

6 timeouted-movingleds

6.1 Presentation

We're gonna now introduce a slight variation on the last example. We want to program, from the start, each time when the leds will "move". Using a timer we can't set more than one alarm (as seting the second one would erase the first). We could use several timer as our AVR provides 6 of them. But we would then be limited to 6. Instead we will add an abstraction to the timer component : a timeout.
Let's have a look at the new architecture :

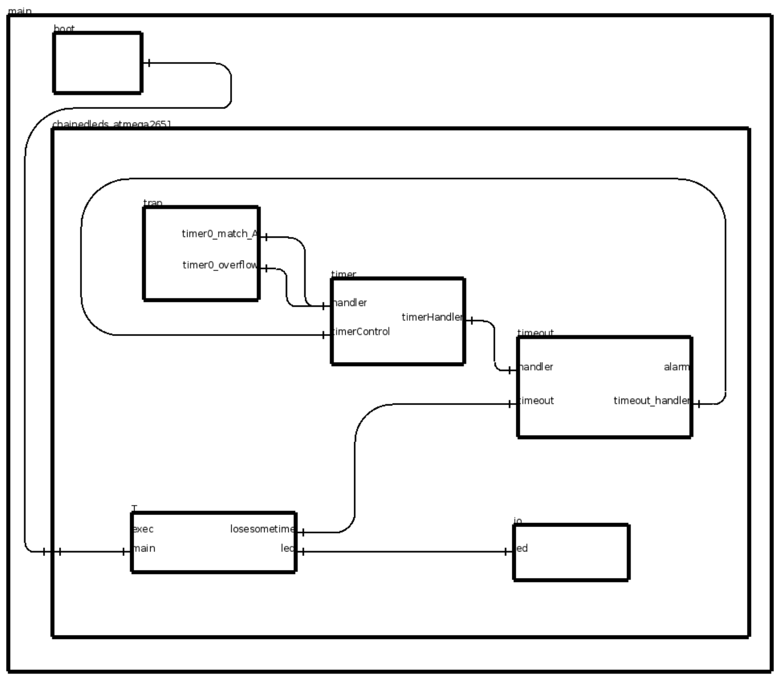


Figure 5: Representation of the timeout_movingleds example ADL.

6.2 Timeout

The role of **timeout** is to maintain a list of all requested alarms, and make sure they get honored. Here come's its IDL :

```
package event.activity.api;

public interface TimeOut {
  void add(unsigned long millisecond, any itf);
}
```

First argument of the **add** method, is the time we want to get the alarm, in millisecond. Second argument, is the interface that should be woken up. This interface as to be a **handler** which as allready been discust on section 4.1
We use this interface in the **main** section of component **T** :

```
void SRV_main__main(jint argc, jstring* argv) {
  jbyte i;
  for (i=1; i<9; i++) {
    CLT_timeout__add( (julong)(i*HOW_MANY_MILLISECONDS), SRVID_exec );
  }
  while(1) { //This is needed if we don't want to stop every component
  }
}
```

The second argument given to the **add** method uses the keyword `SRVID_exec` which will be compiled into a pointer on the **exec** interface of component **T**.

What we really are doing here, is ask for 8 alarms each separated by "`HOW_MANY_MILLISECONDS`" that will all wake up the **exec** interface.

The rest of component **T** hasn't change since last examples.
Let's have a look at the internals of timeout :
The time at which we want the alarm and the interface we will wake up are stored on two arrays that will be used as circular buffers. We keep the position of begining and the end of the data in those buffer with indexes `first_full_slot` and `first_empty_slot`.

```
struct {
  julong times[MAX_NUMBER_OF_TIMER];
  any interfaces[MAX_NUMBER_OF_TIMER];
  jubyte first_full_slot;  //  indice of the next alarm to come.
  jubyte first_empty_slot;  //  indice of the first_empty_slot in the queue
} PRIVATE;
```

6.3 The BindingController interface.

Assuming that we got alarms waiting on the queue; when the component **trap** catch a `match_A` interuption, it then calls the **handler** on the **timer** component which, in turn calls the **handler** on the **timeout** component which then needs to update its tables, ask for an other timer, and wake up the specified interface. Here's the code :

```
static void SRV_handler__execute() {
  jubyte i;
  // If there is no event waiting for an alarm
  // This should never hapen
  if (PRIVATE.first_full_slot == MAX_NUMBER_OF_TIMER) {
    DEBUG_PRINTF("TimeOut buffer is starving !!! \n");
  }
  else {
    // update every alarm time //
    for (i = DECREMENT_POSITION_IN_RING(PRIVATE.first_empty_slot); i != PRIVATE.first_
      PRIVATE.times[i] = PRIVATE.times[i] - PRIVATE.times[PRIVATE.first_full_slot];
    }
    PRIVATE.times[PRIVATE.first_full_slot] = 0;
    //Call the apropriate interface.
    CLT_bc__unbind("timeout_handler");
    CLT_bc__bind("timeout_handler", PRIVATE.interfaces[PRIVATE.first_full_slot]);
    PRIVATE.first_full_slot = INCREMENT_POSITION_IN_RING(PRIVATE.first_full_slot);
    if (PRIVATE.first_full_slot == PRIVATE.first_empty_slot) {
      PRIVATE.first_full_slot = MAX_NUMBER_OF_TIMER; // If the timeout is empty then f
    }
    else {
      CLT_alarm__set(PRIVATE.times[PRIVATE.first_full_slot]); // set the next alarm to
    }
    CLT_timeout_handler__execute(); // execute the event related to the past alarm
  }
}
```

We notice two call on a client interface called **bc**. As seen in the description of **timeout** this interface implements the **BindingController** IDL. It is used to bind the **timeout_handler** interface to the component which required the incoming alarm. It provides 3 methodes :

- `bind(void* clientItfName, void* serverItfId)` which create a binding from the client interface given in the first argument, to the server interface given in the last argument.
- `unbind(void* clientItfName)` which destroy a binding starting from the client interface given as argument.

- `lookup(void* clientItfName)` which returns the server interface bound to the client interface given as argument.

Note : `serverItfId` should be defined by the key word `SRVID_` which as been introduced previously in this section. `cltItfId` should be the name of the interface. Thus it is only possible to -dynamicaly- bind a client interface belonging to the component which provides the **binding controller**.