Dynamic reconfiguration with Nuptse on the Cognichip

Julien TOUS

October 3, 2007

1 introduction

The aim of this document is to present the experiments that were made to dynamicaly reconfigure operating systems written using Nuptse, on the AVR microcontrollers.
For an introduction about using Nuptse you should read the "Think-nuptse-doc" at `https://yourdev.rd.francetelecom.fr/svn/viewvc.php/papers/doc/`. All of the developement explained bellow was made on a AVR microcontroller ATmega2561. Following examples are based on examples explained in "Using Nuptse on AVR" that you can get at `kortex/branches/avr/doc/NuptseOnAVR/`.

2 The reconfigure example (introduction to fractal controllers)

Nuptse (as a Fractal implementation) provides a bunch of controllers we can use to introspect our kernel and modify it's state.

2.1 What the functional part does

The *reconfigure* example is based on the example two_eventscheduled_movingleds. Two "applications" T1 and T2, shares the leds. T1 deals with the first four leds while T2 uses the four last. Both T1 and T2 components are creating "caterpilars" as featured on all movingleds examples.
Component **T1** and **T2** expect to control leds through the interface `SplitedLed`

```
public interface SplitedLed {
   void set4leds(unsigned byte newstate, unsigned byte first_or_last);
}
```

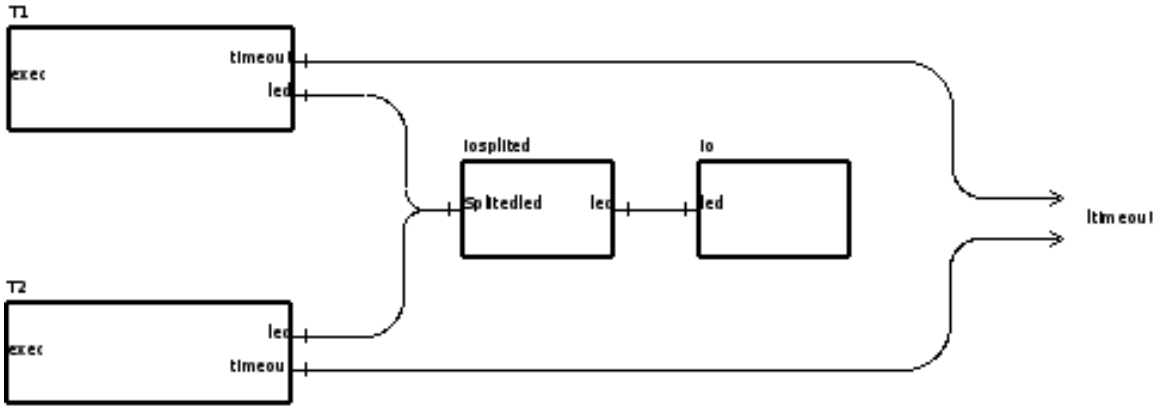

Figure 1: Representation of the applicative part of the *reconfigure* example.

The four least signifiant bit of `newstate` represent four leds (the same way as with the `Led` interface) while `first_or_last` tells which leds are to be used. `0` means first anything else means last.
Here come the `iosplited.c` implementation.

```
void SRV_splitedled__set4leds( jubyte newstate, jubyte first_or_last) {
// Temporary variable for PORTC so intermediate
    // operations aren't visible on the leds
    jubyte oldstate;
oldstate = CLT_led__getLEDs();
DEBUG_PRINTF("io.c : SRV_splitedled__set4leds \n");

if (first_or_last == 0) {
oldstate = oldstate & 240;//
newstate = newstate & 15; //
        DEBUG_PRINTF("first leds are moving ! %u \n", newstate);
}
else {
oldstate = oldstate & 15; //
        newstate = newstate * 16;
newstate = newstate & 240; //
        DEBUG_PRINTF("last leds are moving ! %u \n", newstate);
}

    // Combinates firsts digits off oldstate and lasts off newstate.
oldstate = oldstate | newstate;
    // Apply above calculation on the leds.
CLT_led__setLEDs(oldstate);
}
```

One can see that `iosplited` component requires the `Led` interface. It actually just muxes the two "four leds" representation coming from `T1` and `T2` into a "height leds" representation that get rendered by the `io` component.

2.2 What we want to change

What we want to do is replace components `iosplited` and `io` by a another component `iosplited_reconf` that will, alone, muxes and render the "four leds" representation coming from `T1` and `T2`. Component `iosplited_reconf` should of course implement the `SplitedLed` interface. Here is the code :

```
void SRV_splitedled__set4leds( jubyte newstate, jubyte first_or_last) {
        // Temporary variable for PORTC so that itermediate
        // operations aren't visible on the leds
        jubyte oldstate;

        oldstate = PORTC;
        oldstate = ~oldstate;
        DEBUG_PRINTF("io.c : SRV_led__set4leds \n");

        if (first_or_last == 0) {
                oldstate = oldstate & 240;//
                newstate = newstate & 15; //
        DEBUG_PRINTF("first leds are moving ! %u \n", newstate);
        }
        else {
                oldstate = oldstate & 15; //
                newstate = newstate * 16;
                newstate = newstate & 240; //
                DEBUG_PRINTF("last leds are moving ! %u \n", newstate);
        }

        // Combinates firsts digits off oldstate and lasts off newstate
        oldstate = oldstate | newstate;
        oldstate = ~oldstate;
        PORTC=oldstate; // Apply above calculation on the leds.
}
```

2.3 Global architecture

The code that will execute the reconfiguration is located in component `reconf_handler`. Just after booting the `init` component starts "applications" `T1` and `T2` and calls `init` interface on `reconf_handler`. `init` interface does nothing but ask to be woken up later for reconfiguring.

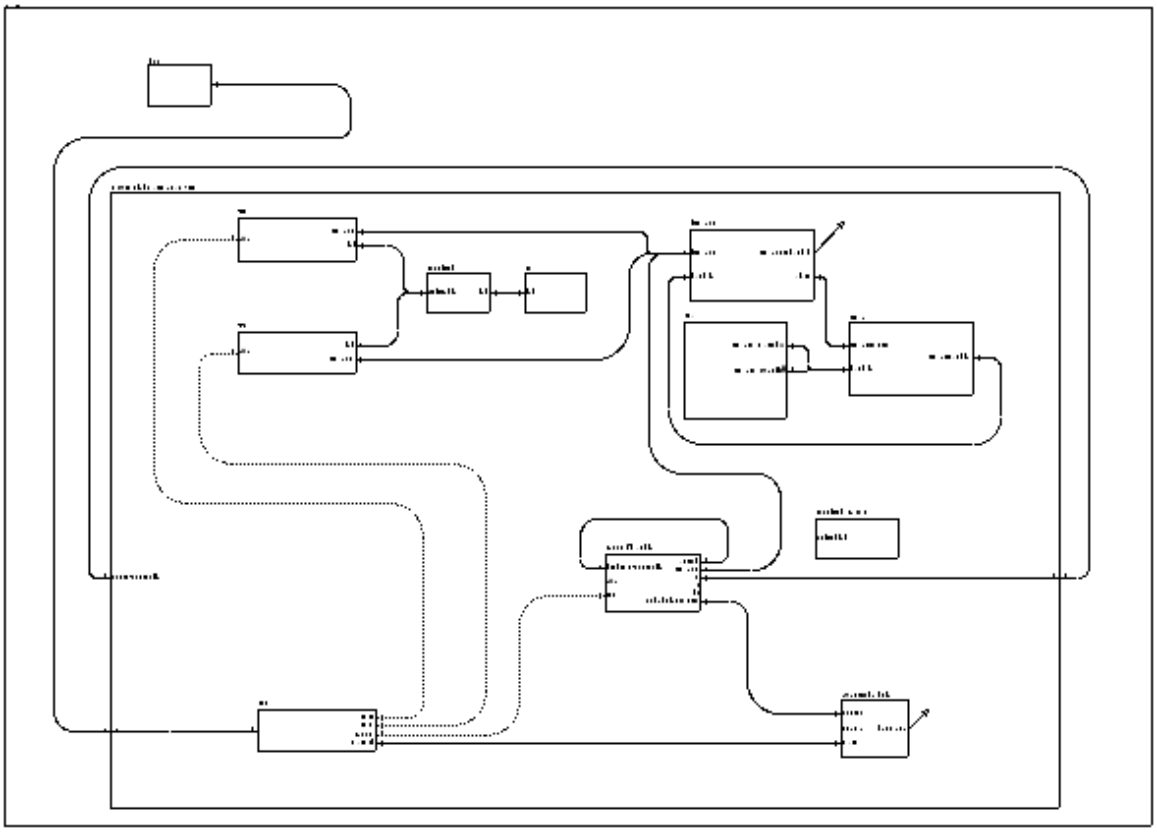

Figure 2: Representation of the ADL.

For now both `reconf_handler` and `iosplited_reconf` are allready part of the kernel at compile time. Compiling and downloading new code will be explained on a later section.
All reconfiguration code is located in `reconf_handler`. It is expected to find interceptors on the bindings from `T1` and `T2` to `iosplited`. We will also intercept binding from `timer` to `timeout` component. Allthought this last interceptor is not truely necessary it is handy to ensure that no dynamic binding and call will be made on `T1` and `T2` while reconfiguring.
The kernel architecture after compilation is represented on figure 3.

2.4 Reconfiguration code

Now that we had an overview of the kernel let's look precisely at what the reconfiguration code does. The simple algorithme is as follow :

- Find `ComponentIdentity` server interface of `iosplited_reconf` through the `ContentController` of the parent component.
- Through the `ComponentIdentity` find the `SplitedLed` server interface.

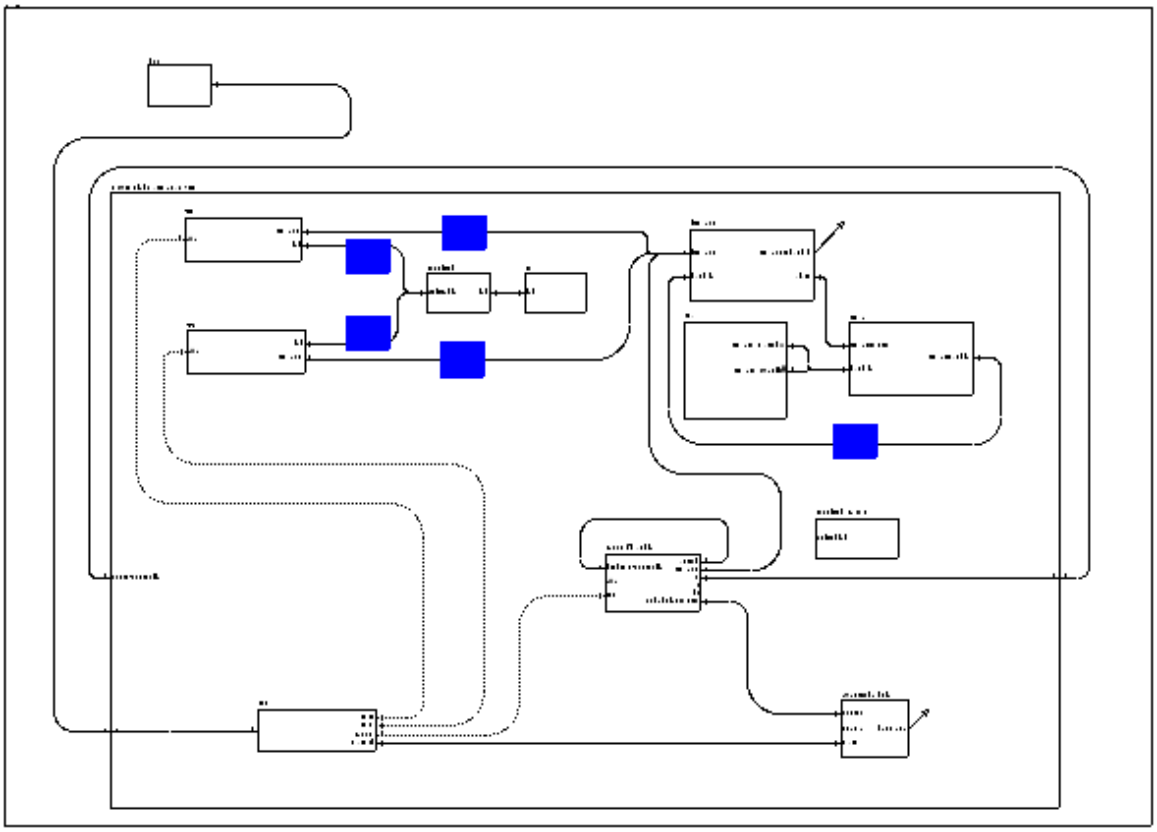

Figure 3: Representation of compiled architecture. Blue components are interceptors created at compile time, and bound to the scheduler.

- For each binding to the old component :
  - Find the `ComponentIdentity` server interface of interceptor using the `ContentController` of the parent component.
  - Through the `ComponentIdentity` find the `BindingController` server interface.
  - Through the `BindingController` bind the `SplitedLed` client interface to the server one on the new component.

This short algorithme is implemented with Nuptse as follow :

```
void SRV_exec__execute() {
    any x_ci;
    any x_bc;
    any srv_itf;

    DEBUG_PRINTF("Welcome in reconf_handler !");
```

```
    // Looking for iosplited_reconf component
    x_ci = CLT_cc__getSubComponent("main.main0.iosplited_reconf");
    // Moving to ComponentIdentity interface
    x_ci = x_ci + sizeof(void *);
    // Binding client interface ci to the previously found server
    CLT_autobc__bind("ci", x_ci);
    // Geting the splitedled interface we want to use now
    srv_itf = CLT_ci__getInterface("splitedled");

    // Looking for interceptor component
    // Beware interceptor numbers can change.
    // Have a look at the compilation output
    x_ci = CLT_cc__getSubComponent("main.main0.interceptor1");
    DEBUG_PRINTF("x_ci %x\n", x_ci);
    // Moving to ComponentIdentity interface
    x_ci = x_ci +sizeof(void *);
    // Binding client interface ci to the previously found server
    CLT_autobc__bind("ci", x_ci);
    // Geting the Binding Controller of the interceptor
    x_bc = CLT_ci__getInterface("binding_controller");
    DEBUG_PRINTF("x_bc %x\n", x_bc);
    // Binding client interface bc to the previously found server
    CLT_autobc__bind("bc", x_bc);
    // Binding the interceptor target to iosplited_reconf component
    CLT_bc__bind("target", srv_itf );
    // Verifying
    DEBUG_PRINTF("target %x \n", CLT_bc__lookup("target"));

    // Same for other interceptor
    x_ci = CLT_cc__getSubComponent("main.main0.interceptor3");
    DEBUG_PRINTF("x_ci %x\n", x_ci);
    x_ci = x_ci + sizeof(void *); //Hack
    CLT_autobc__bind("ci", x_ci);
    x_bc = CLT_ci__getInterface("binding_controller");
    DEBUG_PRINTF("x_bc %x\n", x_bc);
    CLT_autobc__bind("bc", x_bc);
    CLT_bc__bind("target", srv_itf );
    DEBUG_PRINTF("target %x\n", CLT_bc__lookup("target"));
}
```

Note that the component that we do change, doesn't have any client interface. If there would be, we should use a similar algorithme to bind our new component to the clients.

3 The download_and_reconfigure example

Now we know how to find components and interfaces, using fractal introspection, and how to modify bindings. Now we need to learn how to add new code on the running kernel. Dynamicaly linking code seems to be quite unreasonable according to the AVR capabilities. The alternative solution we will describe here is to download staticaly linked code on the AVR. Linkage will be done using GNU Binutils tools on the developement computer.
Compared to the *reconfigure* example, a few thing have changed on the kernel. The functional part is the same, and we will apply the same modification (ie replacing `iosplited` and `io` components with an "all in one" `iosplited_reconf` component). The `iosplited_reconf` and `reconf_handler` components are not part of the initial kernel. Thus we need an infrastucture to download them. We will use the serial port (components `usart` and `serialbuffer`). The architecture of the kernel is represent on figure 4

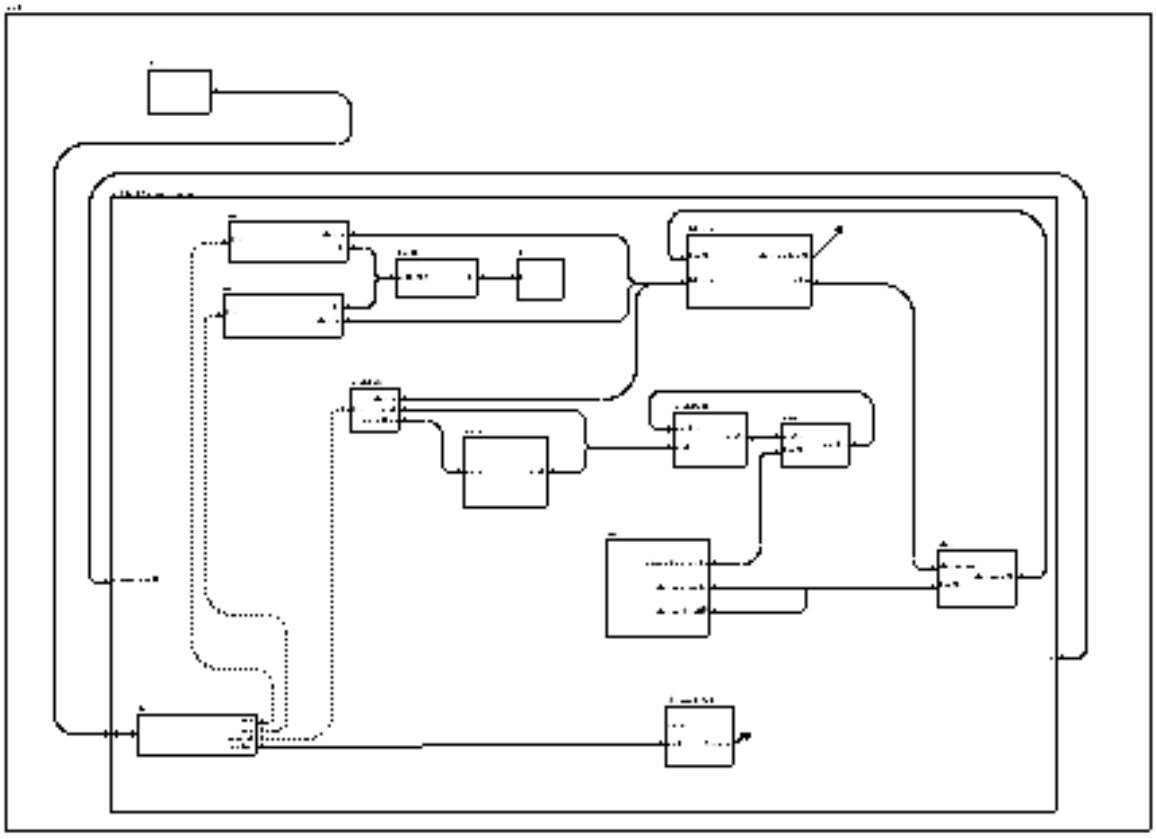

Figure 4: Representation of initial architecture.

3.1 Downloading and writing new code

For those who which to know more about how the serial port works, you should have a look at the (yet to come) radio documentation. We will here talk about the way we use the serial port to download our reconfiguration components and informations.

The `serial_spy` component, is responsible of keeping an eye on the serial buffer. It wakes up every now and then to check if there is any character in the buffer.

- If there is any character but an "R" it sends it back through the serial port.
- If there is an "R", it calls the `reconf` component who will manage to download the code, and write it to flash and RAM.

When `reconf` method on `reconf` component starts, it sends a character to the server. This is a synchronisation message meaning the AVR is ready to receive some data. And then it waits for some reconfiguration parameters. In order :

- The addresse on the flash where it should start to write the `.text` section.
- The size of the `.text` section.
- The addresse on the RAM where it should start to write the `.data` and `.bss` section.
- The size of the concateneted `.data` and `.bss` sections.
- The offset (from the start of the downloaded `.text` section) of the `reconf` method on `reconf_handler` component.
- The `ComponentIdentity` address of the downloaded `iosplited_reconf` component.

Let's have a look at the code which grabs the address of the `.text` section for example.

```
temp = PRV_hex_receive();
text_start_address = temp << 24;
temp = PRV_hex_receive();
text_start_address = text_start_address + temp << 16;
temp = PRV_hex_receive();
text_start_address = text_start_address + temp  << 8;
temp = PRV_hex_receive();
text_start_address = text_start_address + temp;
```

The private function `PRV_hex_receive` is used to grab a binary byte from the serial buffer. `start_text_add` here is of type jlong which is 32 bits wide. As we grab values byte by byte we first store them in a temporary variable and then reconstruct a `jlong`

typed number. First byte is moved 24 bits on the left, the second byte 16, the third 8 and the fourth stays as it is.
The same strategie is used for each of the reconfiguration parameters which are respectily 32, 16, 32, 16, 32 and 32 bits wide.

Actually at compile time you got the choice to download code in ASCII and convert it on the AVR, or to download directly in binary form. Althought the second solution is surely the best, regarding the transfer and treatment time, the first solution is quite convenient for debuging. ASCII transmition is enabled by defining `ASCII_TRANSMITION` with a macroprocessor instruction. If ASCII transmition is enable `PRV_hex_receive` actualy grab two character from the serial buffer and convert then to a single byte numerical value.

```
#ifdef ASCII_TRANSMITION
//recieve 2 ascii char which represent one byte value
jubyte PRV_hex_receive() {
  jubyte c[2];
  jubyte i, val, temp;

  i=0;
  while (i<2) {
       CLT_serial__getchar( 0, &temp);
        if ( (temp > 0x2F) && (temp < 0x3A) ) {
            //If a number (0x30 to 0x39)
            c[i] = temp - 0x30;
            i=i+1;
        }
        else if ( (temp > 0x40) && (temp < 0x47) ) {
            //Uppercase ABCDEF (0x41 to 0x46)
            c[i] = temp - 0x37;
            i=i+1;
        }
        else if ( (temp > 0x60) && (temp < 0x67) ) {
            //Lowercase abcdef (0x61 to 0x66)
            c[i] = temp - 0x57;
            i=i+1;
        }
        else {
    // The value we get was bad.
            // Maybe a control charater.
        }
}
val = (c[0] << 4) + c[1];
```

```
return val;
}
#endif

#ifndef ASCII_TRANSMITION
//recieve a byte
jubyte PRV_hex_receive() {
    jubyte val;

    CLT_serial__getchar( 0, &val);
    return val;
}
#endif
```

Now that every parameter is known, we can start to download the code. Writing on the AVR flash while executing code has a lot of restrictions :

- The code which execute the transfer from RAM to flash must be located in the `.bootloader` section (4 pages starting at 0x3fc00).
- You can't write less than a page (256 bytes) at once.
- Interuption must be disabled while writing.

The first concerne is achieved by specifying the section with the section attribute. Actually the whole `boot_program_page` function, reported below, is discribed in the avr-libc documentation :

```
__attribute__ ((section (".bootloader")))
void boot_program_page(uint32_t  page, uint8_t * buf) {
    uint16_t i;
    uint8_t sreg;

    // Disable interrupts.
    sreg = SREG;
    __asm__ __volatile__("cli");
    eeprom_busy_wait();
    boot_page_erase(page);
    // Wait until the memory is erased.
    boot_spm_busy_wait ();
    for (i=0; i<SPM_PAGESIZE; i+=2) {
        // Set up little-endian word.
        uint16_t w = *(buf++);
        w = w + ( *(buf++) << 8 );
        boot_page_fill(page + i, w);
```

```
    }
    boot_page_write(page);    // Store buffer in flash page.
    boot_spm_busy_wait();     // Wait until the memory is written.
    // Reenable RWW-section again.
    // We need this if we want to jump back
    // to the application after bootloading.
    boot_rww_enable();
    // Re-enable interrupts (if they were ever enabled).
    SREG = sreg;
}
```

The second restriction requires us to start to write the text at the beginning of a page. But as we receive this address as a parameter, this is not a problem here.

The second and third restrictions together forces us to be carefull. While writing to flash interuptions are disabled. All information arriving on serial port would then be lost. To avoid this we simply send data thru' the serial port page by page. When the AVR receives a page it writes it to flash, and then ask for an another one by sending a character. On the other side, the server, sends a page, and then wait for a character to send the next. Here comes the code :

```
    start_text_write = text_start_address;
    // We are waiting for a full page
    while (text_section_size > 256) {
        // Getting a page
        for (PRIVATE.i = 0; PRIVATE.i < 256 ; PRIVATE.i++) {
            data[PRIVATE.i] = PRV_hex_receive();
        }
        j++;
        DEBUG_PRINTF("I: write page %d\n", j);
        // Writting a page to flash
        boot_program_page(start_text_write, data);
        // ajusting information for next page
        text_section_size = text_section_size - 256;
        start_text_write = start_text_write + 256;
        DEBUG_PRINTF("reste %d byte\n", text_section_size);
        // Asking for a new page
        CLT_serial__putchar('Z');
    }

    //The last part doesn't fill a page
    // Getting the end of the code
    for (PRIVATE.i = 0; PRIVATE.i < text_section_size ; PRIVATE.i++) {
```

```
        data[PRIVATE.i] = PRV_hex_receive();
    }
    j++;
    DEBUG_PRINTF("I: write last page, the N:%d \n", j);
    // Writting a page to flash
    boot_program_page(start_text_write, data);
    DEBUG_PRINTF("I: End write flash \n");
    // Asking for the .data section
    CLT_serial__putchar('Z');
```

Althought none of the previous restrictions applyes while writing to RAM, the same strategy is used for downloading the `.data` section.

3.2 Executing the reconfiguration code

Now the code belonging to `reconf_handler` and `iosplited_reconf` components is stored on the flash. We need a way to execute the reconfiguration code. That's the purpose of the `start_offset` parameter we sent in the begining of the transaction. We're gonna assign the address of the `reconf` method of `reconf_handler` to the function pointer `f`. `f` is declared as :

```
void (*f)(any cc, any ci);
```

And here is the code :

```
__asm__ __volatile__("cli");
// Adresses we read using avr-objdump point to 8bit words
// But adresses on the AVR flash point to 16bits words
// Hence instruction at 0xpouet on avr-objdump
// is at (0xpouet / 2) on the AVR
f = (jushort) ((text_start_address + start_offset)/2);
// Jump to the freshly downloaded reconf_handler
f(CC, CI);
__asm__ __volatile__("sei");
```

Assembly instruction `cli` and `sei` respectively disable and and reenable interuptions. As you can see in the comments, there's a kind of magic division to do before callping the reconf function. A beter explanation than that i could give can be found, on the GDB manual *"Debugging with gdb"* by *Richard Stallman, Roland Pesch, Stan Shebs, et al.* at chapter 9.4.

The `reconf_handler` component implements the same algorhithm as in the *reconfigure* example. The only notable differences being that `reconf_handler` isn't bound to the parent `ContentController` and that the parent `ContentController` isn't aware of the `iosplited_reconf` component. That's why we pass them as arguments of the `reconf` method. Here is `reconf` interface IDL :

```
package avr.boot.api;

interface reconf {
  void reconf(any CC, any CI);
}
```

Any experienced Nuptse user should be alarmed by the fact that the Nuptse compiler uses to add an extra argument to every interface method. Thus the function call `f(CC, CI);` should not match the compiled definition of the `reconf` method. Actually the Nuptse compiler allows to leave some method untouched by declaring then `single` :

```
component reconf_handler {
provides avr.boot.api.reconf as reconf [single=true]
requires fractal.api.BindingController as bc
requires fractal.api.ContentController as cc
requires fractal.api.ComponentIdentity as ci

content avr.reconf__handler
}
```

Beware that declaring a component or interface `single`, prevents you from using two instances of this component.

4 Server side operations

We saw that to complete the reconfiguration, the AVR expect data to come "formated". Here we will describe the operations done on the server to execute the reconfiguration. Those operations have arbitrary been divided into three steps :

- Compiling the new code we will inject on the AVR as ELF objects.
- Staticaly link this code to the running kernel, create binary objects and reconfiguration parameters.
- Send all required data to the AVR, folowing the simple synchronisation protocole.

4.1 Step 1 : Creating ELFs

To create the ELFs objects we simply compile a kernel containing the reconfiguration code and the new componant. The `build.xml` file contains two targets : `avr` for the initial kernel, `avr-reconf` for the fake reconfigured kernel. The (unusefull) fake reconfigured kernel is written as `Rkernel`, and all coresponding objects start with an "R". The objects file of interest for us are : `build/obj_atm2561/Rmain_main0_reconf_handler_*` and `build/obj_atm2561/Rmain_main0_iosplited_reconf_*`.

4.2 Step 2 : Creating binary and parameters

The `./reconf.sh` Bash script will take care of creating all the binary and parameters we will send. The first argument we should pass to `./reconf.sh` is the running kernel ELF file. Then all ELFs object file corresponding to the code we will upload.
For our example :

```
$ ./reconf.sh kernel build/obj_atm2561/Rmain_main0_reconf_handler_*
build/obj_atm2561/Rmain_main0_iosplited_reconf_*
```

Let's now have a look at `./reconf.sh` script which is pretty much self explained :

```
#!/bin/bash
#name of the running kernel elf file
kernel=$1

#All the files created by this script will start with "new_component"
output_objfile=new_component
#Removing reconfiguration files from previous run
rm new_component*

#Looking how much flash we allready use on the cognichip.
start_text_add=‘avr-size $kernel \
                | grep $kernel \
                | awk ’{print $5}’‘

#We will start to write new_component .text section at the begining
#of a page (256Byte or 0x100 upper approximation)
start_text_add=$(( ((0x$start_text_add + 0x100)/0x100)*0x100 ))

#Convert it to hexadecimal base
start_text_add=‘echo "obase=16; $start_text_add" | bc‘
echo start_text_add $start_text_add

#Looking how much RAM we allready use on the cognichip.
size_bss=‘avr-objdump  -h kernel \
          | grep .bss \
          | awk ’{print $3}’‘
begining_bss=‘avr-objdump -h kernel \
              | grep .bss \
              | awk ’{print $4}’‘

#We will start to write new_component .data and
#.bss sections just after current .bss
```

```
start_data_add=$(( 0x$begining_bss + 0x$size_bss ))

#Convert it to hexadecimal base
start_data_add=‘echo "obase=16; $start_data_add" | bc‘
echo start_data_add $start_data_add

echo "I: create object file with resolved symboles and relative addressing"
  #Linking reconfiguration object files using currently running kernel symbol.
  avr-ld -noinhibit-exec \
         -Ttext $start_text_add \
         -Tdata $start_data_add \
         -o $output_objfile \
         -R $*
  # -noinhibit-exec tells to output a file even if some symbols are unresolved.
  # -Tsection tells at which address should the section start.
  # $* represent all arguments
  # There is a distinction between $1 and all others.
  # Argument $1 is the kernel to reconfigure on which -R option applyes.
  # All others arguments are objects to link.

  #new_component ELF file is now created.
  #When the reconfiguration is finished you can use
  #add-sym nem_component in gdb to debug the new code.

# getting new component "component identity" address
#Warning iosplited_reconf_compDesc should be renamed to match your case
  CI_address=‘avr-objdump -D $output_objfile \
          | grep iosplited_reconf_compDesc \
          | awk ’{print $1}’‘
  echo CI_address $CI_address

# getting new component reconfiguration code "SRV_reconf__reconf" address
  reconfigure_address=‘avr-objdump -D $output_objfile \
                   | grep reconf__reconf \
                   | awk ’{print $1}’‘
  #We will actually send the offset from
  #the start of new_component to reconf method.
  #There is no technical reason to do so. Feel free to change that !
  reconfigure_offset=$((0x$reconfigure_address - 0x$start_text_add))
  #Convert it to hexadecimal base
  reconfigure_offset=‘echo "obase=16; $reconfigure_offset" | bc‘

echo "I: create reconfiguration bin"
```

```
  #extracting .text section in binary format
  avr-objcopy -j .text -O binary  $output_objfile $output_objfile.text.bin

  #extracting .data an .bss section in binary format
  avr-objcopy -j .data -O binary  $output_objfile $output_objfile.data.bin
  avr-objcopy -j .bss  -O binary  $output_objfile $output_objfile.bss.bin
  #We don't need to treat bss and data differently
  #so let's make one file with both.
  cat $output_objfile.bss.bin >> $output_objfile.data.bin

#It's a bit easier to debug transmition from the PC to the cognichip if
#we send ASCII instead of binary.
#We will create both. It's up to you to chose the one you'll send.
#We're now converting binary to ASCII representation.
echo "I: create reconfiguration ascii"
  od -A n -t xC $output_objfile.text.bin \
   | tr -d ' ' \
   | tr -d [:cntrl:] \
   > $output_objfile.text.ascii
  od -A n -t xC $output_objfile.data.bin \
   | tr -d ' ' \
   | tr -d [:cntrl:] \
   > $output_objfile.data.ascii
  od -A n -t xC $output_objfile.bss.bin \
   | tr -d ' ' \
   | tr -d [:cntrl:] > $output_objfile.bss.ascii
  #cat bss to add bss to data.
  cat $output_objfile.bss.ascii >> $output_objfile.data.ascii

echo "I: get stat info"
  #getting the size of .text section
  taille_section_text_ascii=`stat -c "%s" $output_objfile.text.ascii`
  taille_section_text=$(($taille_section_text_ascii / 2))
  echo "I: taille_section_text: $taille_section_text"
  #getting the size of .data and .bss section
  taille_section_data_ascii=`stat -c "%s" $output_objfile.data.ascii`
  taille_section_data=$(($taille_section_data_ascii / 2))
  echo "I: taille_section_data: $taille_section_data"

echo "I: write reconfig info to files"
# -ne options are usefull to avoid having control charateres
# printf format is important as we will count characters on the cognichip
  echo -ne text_start
```

```
  printf "%0.8x \n" 0x$start_text_add
  printf "%0.8x" 0x$start_text_add > $output_objfile.text_start
  echo -ne text_size
  printf "%0.4x \n" $taille_section_text
  printf "%0.4x" $taille_section_text > $output_objfile.text_size
  echo -ne data_start
  printf "%0.8x \n" 0x$start_data_add
  printf "%0.8x" 0x$start_data_add > $output_objfile.data_start
  echo -ne data_size
  printf "%0.4x \n" $taille_section_data
  printf "%0.4x" $taille_section_data > $output_objfile.data_size
  echo -ne reconf_offset
  printf "%0.8x \n" 0x$reconfigure_offset
  printf "%0.8x" 0x$reconfigure_offset > $output_objfile.reconf_offset
  echo -ne CI_address
  printf "%0.8x \n" 0x$CI_address
  printf "%0.8x" 0x$CI_address > $output_objfile.CI_address

#Create a big ASCII file with all reconfiguration info
cat $output_objfile.text_start \
    $output_objfile.text_size \
    $output_objfile.data_start \
    $output_objfile.data_size \
    $output_objfile.reconf_offset \
    $output_objfile.CI_address \
    > $output_objfile.reconf_parameters.ascii

#Create a big binary file with all reconfiguration info
  xxd -r -p -g 0 $output_objfile.reconf_parameters.ascii \
      $output_objfile.reconf_parameters.bin


#Cut the ascii files we will send into 512Bytes pieces
#512Bytes of ASCII gives 256Bytes of binary code on the cognichip.
NbToWrite=512
echo "I: create text section in ascii"
  writed=0
  N=0
  while [ $writed -lt $taille_section_text_ascii ]
  do
    echo "I: create .text part $N"
    dd if=$output_objfile.text.ascii \
       of=$output_objfile.text.ascii.$N \
```

```
       bs=$NbToWrite count=1 skip=$N
    N=$(($N + 1))
    writed=$(($writed + $NbToWrite))
  done
  M=$N
echo "I: create data section in ascii"
  writed=0
  while [ $writed -lt $taille_section_data_ascii ]
  do
    echo "I: create .data part $N"
    dd if=$output_objfile.data.ascii \
       of=$output_objfile.data.ascii.$N \
       bs=$NbToWrite count=1 skip=$(($N-$M))
    N=$(($N + 1))
    writed=$(($writed + $NbToWrite))
  done

#Cut the binary files we will send into 256Bytes pieces.
NbToWrite=256
echo "I: create text section in binary format"
  writed=0
  N=0
  while [ $writed -lt $taille_section_text ]
  do
    echo "I: create .text part $N"
    dd if=$output_objfile.text.bin of=$output_objfile.text.bin.$N bs=$NbToWrite count=
N=$(($N + 1))
writed=$(($writed + $NbToWrite))
done
M=$N
echo "I: create data section in binary format"
writed=0
while [ $writed -lt $taille_section_data ]
do
echo "I: create .data part $N"
    dd if=$output_objfile.data.bin \
       of=$output_objfile.data.bin.$N \
       bs=$NbToWrite count=1 skip=$(($N-$M))
    N=$(($N + 1))
    writed=$(($writed + $NbToWrite))
  done

echo
```

```
echo "To execute reconfiguration, run : "
echo "./upload_ASCII_reconf.py"
echo "or"
echo "./upload_binary_reconf.py"

exit
```

4.3 Step 3 : Sending everything on the AVR

You can chose from two scripts to execute reconfiguration :

- One for ASCII transmition : `upload_ASCII_reconf.py`.
- One for binary transmition : `upload_binary_reconf.py`.

Make sure you use ASCII transmition if you defined `ASCII_TRANSMITION` in `reconf` component, and binary transmition otherwise.
Those scripts simply initialise the serial port on the server, and follow the following protocol for uploading reconfiguration parameters and code. They require no arguments.

- Send an "R" to intialise reconfiguration.
- Wait for one character.
- Send reconfiguration parameters.
- Wait for one character.
- For each "text" and "data" file created previously :
  - Send the file.
  - Wait for one character.

4.4 Step 4

Look at the leds !