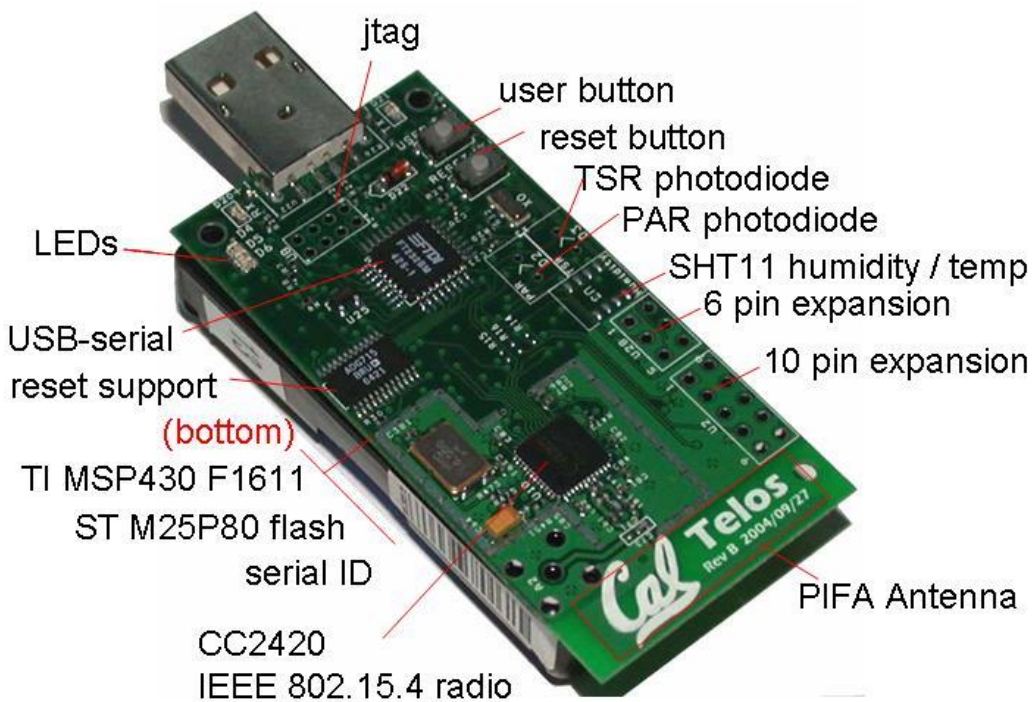
Programmation de motes en NesC

Alexandre Guitton

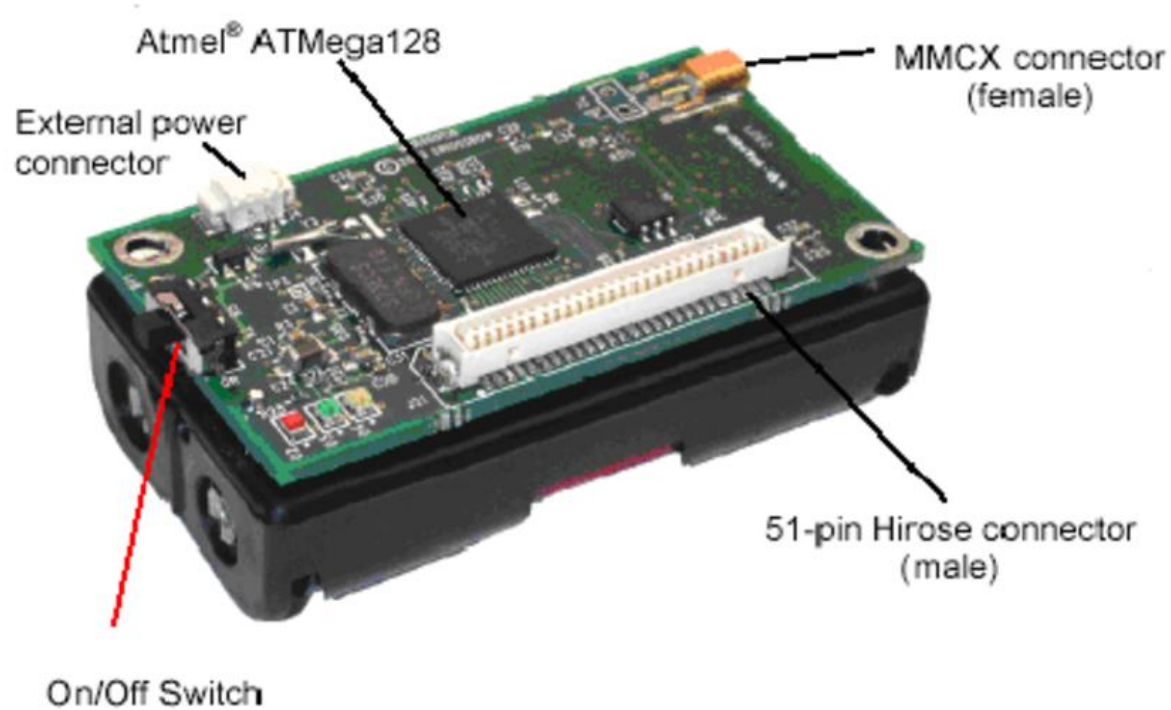
Introduction

- Définitions
  - Mote = nœud d'un réseau de capteurs (exemple : TelosB, MICA2, Arduino)
  - NesC = langage de programmation basé sur le C
  - TinyOS = système d'exploitation des motes (parmi d'autres, comme Contiki ou FreeRTOS)
- Pourquoi un nouvel OS et un nouveau langage ?
  - Les motes ont peu de ressources (mémoire, puissance de calcul, énergie) => OS simple, bibliothèque simplifiée
  - Les motes sont spécifiques => simplicité pour la gestion des capteurs / actionneurs

Motes (1/2)



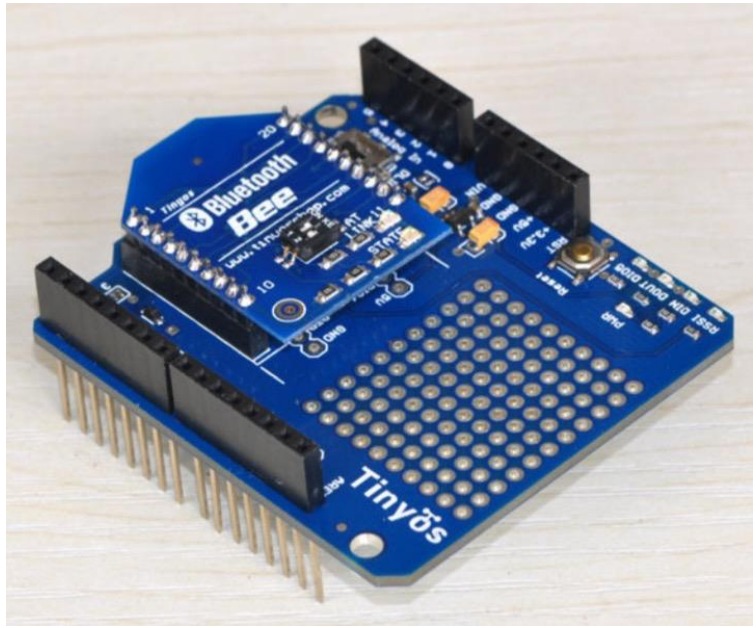
TelosB



MICA2

Motes (2/2)

Uno (Arduino)



Arduino

Waspmote
(Arduino)

TinyOS

- OS développé par Berkeley pour leurs motes, en open source
  - Programmé en NesC
- Une seule pile partagée par toutes les applications
- Gestion de la mémoire limitée, gestion des processus limitée (ordonnanceur FIFO, avec un seul niveau de préemption)

NesC

- Une application en NesC est constituée :
  - D'un ensemble de composants
  - Ces composants sont connectés entre eux via un graphe statique (décrit dans un fichier de configuration)
  - Ces composants exécutent des actions, qui sont soit des événements, soit des tâches
- Un événement correspond à une action non interruptible (donc courte), déclenchée suite à un stimulus (timer qui expire, donnée collectée, etc.)
- Une tâche correspond à une action interruptible par un événement (mais pas par une autre tâche)

Composants

- Les composants implémentent des commandes et utilisent des événements
  - Les commandes peuvent poster (= créer) des tâches mais pas lancer d'événements
  - Ils sont décrits dans un fichier d'interface, et sont implémentés dans un fichier module

Ordonnanceur

- L’ordonnanceur est un simple FIFO
- Les tâches peuvent être préemptées par des événements, mais ne peuvent pas être préemptées entre-elles
- Les événements peuvent être préemptés par d’autres événements ou par des interruptions matérielles

Mots-clés du langage

- Un bloc d’instructions peut être rendu atomique avec le mot-clef « atomic »
  - Les accès concurrents sont évités, soit en les mettant dans des tâches, soit en les accédant via des blocs atomiques
- Pour poster une tâche : « post »
- Pour appeler une commande : « call »
- Pour lancer un événement : « signal »

Exemple 1 : le programme Blink

(remarque : certains exemples nécessitent TinyOS 1, d'autres TinyOS 2 ☹)

Exemple 1 : le programme Blink

- Fichier de configuration Blink.nc
- Implémentation
  - Utilise quatre composants : Main (pour être exécutable), le module BlinkM, SingleTimer, et LedsC
  - Connecte le contrôle de Main à BlinkM et à SingleTimer
  - Connecte le timer de Blink à SingleTimer.Timer
  - Connecte les LED de BlinkM à LedsC.Leds

```
configuration Blink {
}

implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> BlinkM.StdControl;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

Module BlinkM

- Composant principal de notre application
  - Fournit (= implémente) l'interface StdControl (fonctions init, start et stop)
  - => donc chaque commande de l'interface StdControl apparaît dans l'implémentation, et les événements peuvent être lancés
  - Utilise les interfaces Timer et Leds
  - => donc chaque événement de ces interfaces est implémenté, et les fonctions peuvent être appelées

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }
  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 1000) ;
  }
  command result_t StdControl.stop() {
    return call Timer.stop();
  }
  event result_t Timer.fired() {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

Interface StdControl

- L'interface StdControl déclare trois commandes (init, start et stop) qui retournent un booléen, et pas d'événement
- Remarques :
  - L'initialisation d'un composant doit être faite avant son démarrage
  - L'initialisation d'un composant doit initialiser tous les composants qu'il utilise

```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

Interface Timer

- Déclare deux commandes (start et stop) et un événement
- La commande start peut correspondre soit à un déclenchement répétitif (TIMER_REPEAT), soit à un déclenchement unique (TIMER_ONE_SHOT)
- Pour les composants qui utilisent cette interface (dont BlinkM), start et stop peuvent être appelées, fired doit être implémenté
- Pour les composants qui fournissent cette interface (dont SingleTimer), star et stop doivent être implémentés, fired peut (= devrait) être appelé

```
interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}
```

Interface Leds

- Pour les composants qui utilisent cette interface, toutes ces commandes peuvent être appelées, et tous les événements doivent être implémentés (mais il n'y en a pas)

```
#include "Leds.h"

interface Leds {
  command void led0On();
  command void led0Off();
  command void led0Toggle();
  command void led1On();
  command void led1Off();
  command void led1Toggle();
  command void led2On();
  command void led2Off();
  command void led2Toggle();
  command uint8_t get();
  command void set(uint8_t val);
}
```

Composant LedsC

- Ce composant est une application…
  - … qui fournit l'interface Leds
  - … qui utilise deux autres composants (LedsP et PlatformLedsC)
  - … qui se contente de connecter les sous-composants entre eux (et à l'interface Leds)

```
configuration LedsC {
  provides interface Leds;
}
implementation {
  components LedsP, PlatformLedsC;
  Leds = LedsP;
  LedsP.Init <- PlatformLedsC.Init;
  LedsP.Led0 -> PlatformLedsC.Led0;
  LedsP.Led1 -> PlatformLedsC.Led1;
  LedsP.Led2 -> PlatformLedsC.Led2;
}
```

Composant PlatformLedsC

- Ce composant est aussi une application…
  - … qui fournit plusieurs interfaces (Led0, Led1, Led2)
  - … qui utilise plusieurs composants qui dépendent de la plateforme (TelosA ici)

```
#include "hardware.h"
configuration PlatformLedsC {
  provides interface GeneralIO as Led0;
  provides interface GeneralIO as Led1;
  provides interface GeneralIO as Led2;
  uses interface Init;
}
implementation {
  components
     HplMsp430GeneralIOC as GeneralIOC,
     new Msp430GpioC() as Led0Impl,
     new Msp430GpioC() as Led1Impl,
     new Msp430GpioC() as Led2Impl;
  components PlatformP;
  Init = PlatformP.LedsInit;
  Led0 = Led0Impl;
  Led0Impl -> GeneralIOC.Port54;
  Led1 = Led1Impl;
  Led1Impl -> GeneralIOC.Port55;
  Led2 = Led2Impl;
  Led2Impl -> GeneralIOC.Port56;
}
```

Composant Main

- Ce composant est aussi une application…
  - … qui utilise l'interface Init
  - … qui fournit l'interface Boot

```
#include "hardware.h"
configuration Main {
  provides interface Boot;
  uses interface Init as SoftwareInit;
}
implementation {
  components PlatformC, RealMainP, TinySchedulerC;
  RealMainP.Scheduler -> TinySchedulerC;
  RealMainP.PlatformInit -> PlatformC;
  SoftwareInit = RealMainP.SoftwareInit;
  Boot = RealMainP;
}
```

Composant SingleTimer

- Ce composant est une instanciation d'un autre composant, nommé TimerC (paramétré par un singleton)

```
configuration SingleTimer {
  provides interface Timer;
  provides interface StdControl;
}

implementation {
  components TimerC;
  Timer = TimerC.Timer[unique("Timer")];
  StdControl = TimerC;
}
```

Flashage du mote

- Commande :
  - make mica install

```
  compiling Blink to a mica binary
  ncc -board=micasb -o build/mica/main.exe -Os -target=mica  -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -finline-limit=200
-fnesc-cfile=build/mica/app.c  Blink.nc -lm
  avr-objcopy --output-target=srec build/mica/main.exe
  build/mica/main.srec
      compiled Blink to build/mica/main.srec


      installing mica binary
  uisp -dprog=dapa  --erase
  pulse
  Atmel AVR ATmega128 is found.
  Erasing device ...
  pulse
  Reinitializing device
  Atmel AVR ATmega128 is found.
  sleep 1
  uisp -dprog=dapa  --upload if=build/mica/main.srec
  pulse
  Atmel AVR ATmega128 is found.
  Uploading: flash
  sleep 1
  uisp -dprog=dapa  --verify if=build/mica/main.srec
  pulse
  Atmel AVR ATmega128 is found.
  Verifying: flash
```

Exemple 2 : le programme Sense

Sense (1/2)

```
configuration SenseAppC {
}
implementation {
  components SenseC, MainC, LedsC, new TimerMilliC();
  components new DemoSensorC() as Sensor;
  SenseC.Boot -> MainC;
  SenseC.Leds -> LedsC;
  SenseC.Timer -> TimerMilliC;
  SenseC.Read -> Sensor;
}
```

```
module SenseC
{
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}
implementation {
  …
}
```

Sense (2/2)

```
#define SAMPLING_FREQUENCY 100
 event void Boot.booted() {
   call Timer.startPeriodic(SAMPLING_FREQUENCY);
 }
 event void Timer.fired() {
   call Read.read();
 }
 event void Read.readDone(error_t result, uint16_t data) {
   if (result == SUCCESS){
     if (data & 0x0004) call Leds.led2On();
     else call Leds.led2Off();
     if (data & 0x0002) call Leds.led1On();
     else call Leds.led1Off();
     if (data & 0x0001) call Leds.led0On();
     else call Leds.led0Off();
   }
 }
}
```

Sense – avec une sauvegarde des données

```
event result_t ADC.dataReady(uint16_t data) {
  putdata(data);
  post processData();
  return SUCCESS;
}

task void processData() {
  int16_t i, sum=0;
  atomic {
    for (i=0; i < size; i++)
      sum += (rdata[i] >> 7);
  }
 // TODO : do something with « sum »
}
```

Exemple 3 : le programme Radio

Radio (1/4) – le fichier d'entête (.h)

```
#ifndef BLINKTORADIO_H
#define BLINKTORADIO_H

enum {
  AM_BLINKTORADIO = 100, // type de paquet (arbitraire)
  TIMER_PERIOD_MILLI = 250
};

typedef nx_struct BlinkToRadioMsg {
  nx_uint16_t nodeid;
  nx_uint16_t counter;
} BlinkToRadioMsg;

#endif
```

Radio (2/4) – le fichier de configuration

```
#include <Timer.h>
#include "BlinkToRadio.h"

configuration BlinkToRadioAppC {
}
implementation {
  components MainC;
  components LedsC;
  components RadioC as App;
  components new TimerMilliC() as Timer0;
  components ActiveMessageC;
  components new AMSenderC(AM_BLINKTORADIO); // to send
  components new AMReceiverC(AM_BLINKTORADIO); // to receive
  App.Boot -> MainC;
  App.Leds -> LedsC;
  App.Timer0 -> Timer0;
  App.Packet -> AMSenderC;
  App.AMPacket -> AMSenderC;
  App.AMSend -> AMSenderC;
  App.AMControl -> ActiveMessageC;
  App.Receive -> AMReceiverC;
}
```

Radio (3/4) – le fichier de module

```
#include <Timer.h>
#include "BlinkToRadio.h"

module BlinkToRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
// to send
  uses interface Packet;
  uses interface AMPacket;
  uses interface AMSend;
  uses interface SplitControl as AMControl;
// to receive
  uses interface Receive;
}
implementation {
  uint16_t counter = 0;
  bool busy = FALSE;
  message_t pkt;
```

```
event void Boot.booted() {
  call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
}
event void Boot.booted() {
  call AMControl.start();
}
event void AMControl.startDone(error_t err) {
  if (err == SUCCESS) {
    call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
  }
  else {
    call AMControl.start();
  }
}
event void AMControl.stopDone(error_t err) {
}
```

Radio (4/4) – le fichier de module

```
 event void Timer0.fired() {
    counter++;
    call Leds.set(counter);
    if (!busy) {
      BlinkToRadioMsg* btrpkt =
(BlinkToRadioMsg*)(call
Packet.getPayload(&pkt, sizeof
(BlinkToRadioMsg)));
      btrpkt->nodeid = TOS_NODE_ID;
      btrpkt->counter = counter;
      if (call
AMSend.send(AM_BROADCAST_ADDR,
&pkt, sizeof(BlinkToRadioMsg)) ==
SUCCESS) {
        busy = TRUE;
      }
    }
  }
```

```
 event void AMSend.sendDone(message_t* msg, error_t error) {
    if (&pkt == msg) {
      busy = FALSE;
    }
  }
  event message_t* Receive.receive(message_t* msg, void* payload,
uint8_t len) {
    if (len == sizeof(BlinkToRadioMsg)) {
      BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)payload;
      call Leds.set(btrpkt->counter);
    }
    return msg;
  }
}
```