Mote-mote rádiós kommunikáció

Feladat egy olyan alkalmazás készítése, amely a mote-on bizonyos időközönként növel egy számláló értékét, majd az új számláló értéket elküldi a rádión keresztül. Az üzenetet fogadó mote megjeleníti a számláló alsó 3 bitjének értékét a led-eken. Ez a folyamat természetesen visszafele is működik, azaz a fogadó mote is periódikusan növel egy számláló értéket, és elküldi azt a rádión keresztül a szomszédjainak.

Első lépésként készítsünk egy olyan alkalmazást, amely bizonyos időközönként megnöveli a számláló értékét, és a számláló alsó három bit-jét megjeleníti a led-eken. Hozzunk létre először is egy *BlinkToRadioC.nc* file-t. Ez a file lesz az alkalmazásunk konfigurációs file-ja. A file-ba a következőket írjuk be:

```
configuration BlinkToRadioC {
 }
 implementation {
   components MainC;
   components LedsC;
   components BlinkToRadioP as App;
   components new TimerMilliC() as Timer0;

   App.Boot -> MainC;
   App.Leds -> LedsC;
   App.Timer0 -> Timer0;
 }
```

Először is szükségünk van egy *MainC* komponensre, ami a *Boot* interface-t biztosítja. Ez az interface tartalmazza a *Boot.booted()* event-et, mely az összes alkalmazás belépési pontja. Ez jelzi a felhasználónak, hogy a mote elindult, és az alkalmazásunk elkezdheti a működését. Szükségünk van még egy *LedsC* komponensre, mely a led-ek kezeléséhez szükséges *Leds* interface-t biztosítja. A *TimerMilliC()* komponens, a mote-on lévő mikrokontrollerben található időzítők kezeléséhet szükséges *Timer<>* generikus interface-t biztosítja. Huzalozzuk össze ezeket a komponenseket a *BlinkToRadioP* modullal, mely az alkalmazásunkat fogja tartalmazni, és itt *App*-nak neveztünk át.

Második lépésként hozzuk létre a *BlinkToRadioP.nc* file-t. Ez a file fogja magát az alkalmazásunkat tartalmazni. A file-ba írjuk be a következőket:

```
module BlinkToRadioP {
   uses interface Boot;
   uses interface Leds;
   uses interface Timer<TMilli>;
 }
 implementation {
   uint16_t counter = 0;

   event void Boot.booted() {
     call Timer.startPeriodic(5000);
   }

   event void Timer.fired() {
     counter++;
     call Leds.set(counter);
   }
 }
```

A *BlinkToRadioP* modul használja, a *Boot*, a *Leds*, és a *Timer<TMilli>* interface.-eket. Ezek az interface-ek megtalálhatóak a *(TOSROOT)/tos/interfaces/*, illeve a *(TOSROOT)/tos/lib/timer* mappákban. Ezeket már be is kötöttük a konfigurációs file-ba a megfelelő komponensekhez. Definiáljunk egy *counter* változót, ennek ez értékét fogjuk periódikusan növelni. Ha a rendszer elindult, azaz a *Boot.booted()* event generálódott, akkor

elindítunk egy időzítőt a *Timer.startPeriodic(5000)* paranccsal, periódikus üzemmódban, úgy hogy 5000 millisecunum-onként generálódjon egy *Timer.fired()* event. Ha ez az event bekövetkezik, akkor a *counter* értékét növeljük, illetve a *Leds.set()* paranccsal a három led-et bekapcsoljuk az új *counter* érték alsó három bit-jének megfelelően.

Harmadik lépésként készítsük el a *Makefile*-t, mely segítségével le lehet fordítani az alkalmazásunkat. A *Makefie* a következőeket tartalmazza:

```
COMPONENT=BlinkToRadioC
include $(MAKERULES)
```

Ezután fordítsuk le és programozzuk fel az alkalmazásunkat a mote-ra a következő képen:

```
make install telosb
```

Negyedik lépésként egészítsük ki a kódunkat, úgy hogy a *counter* tartalmát küldje el rádión, és ha egy rádiós üzenet érkezik, akkor az abban található *counter* értékének alsó három bit-jét jelenítse meg a Led-eken.

A TinyOS-ben az üzenetküldés egy *message_t* struktúrán keresztül történik. A *message_t* struktúra, az alábbi módon néz ki:

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_footer_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Az üzenet elküldéséhez az elküldendő üzenetet a *data* mezőbe fogjuk beírni, azaz ide írjuk be a *counter* értékét. A *TOS_DATA_LENGTH* adja meg a *data* terület méretét, ennek mérete 28 byte, de növelhető egészen 144 byte-ig. Annak érdekében, hogy el tudjuk küldeni a *counter* értélét készítenünk kell egy üzenet struktúrát, mely megmondja, hogy a rádiós üzenet *data* mezőjében milyen elrendezésben vannak elhelyezve az egyes byte-ok. Pl.: a 6 byte az 6 uint8_t vagy 3 uint16_t stb…. A struktúra elkészítéséhez készítsünk egy *BlinkToRadio.h* header file-t, melynek a tartalma legyen a következő:

```
#ifndef BLINKTORADIO_H
#define BLINKTORADIO_H

typedef nx_struct BlinkToRadioMsg {
  nx_uint16_t counter;
} BlinkToRadioMsg;

#endif
```

Ebben a file-ban definiálunk egy struktúrát. A struktúra létrehozásában nincs semmi rendhagyó, kivéve a *struct* és az *uint16_t* kulcsszavakat, melyeket megelőz egy *nx_* kulcsszó. Ez a nesC-ben azt jelenti, hogy az értékek big endian elrendezésűek, míg az *nxle_* azt jelenti, hogy az értékek little endian elrendezésűek. Ezzel el lehet kerülni a különböző platformok közötti endianitási különbözőségeket.

Ötödik lépésként nézzük meg, hogy milyen *interface*-ek is kellenek annak érdekében, hogy el tudjuk küldeni az üzenetünket. Kell az *AMSend* és a *SplitControl* interface. Az *AMSend* olyan parancsokat tartalmaz, amivel az üzenetünket bele lehet tölteni a *message_t* struktúra *data* (*payload*) területére, illetve amelyek segítségével el lehet küldeni az üzenetet a rádión keresztül. A *SplitControl* interface pedig olyan parancsokat tartalmaz, amivel be lehet kapcsolni a rádiót. Ezek az interface-ek megtalálhatóak a *(TOSROOT)/opt/interfaces* mappában. A *BlinkToRadioP* modulunkat ennek megfelelően egészítsük ki az *AMSend* és a *SplitControl* interface-el.

```
module BlinkToRadioP {
  ...
  uses interface AMSend;
  uses interface SplitControl;
}
```

Deklaráljunk két változót:

```
...
implementation {
  bool busy = FALSE;
  message_t pkt;
  ...
}
```

Annak érdekében, hogy használni tudjuk, a rádió-t be kell kapcsoljuk. Ezt a *SplitControl.start()* paranccsal tehetjük meg. A rádió bekapcsolódásáról pedig a *SplitControl.startDone()* event tájékoztat minket. Ennek megfelelően a következő képen egészítsük ki a kódot:

```
...
event void Boot.booted() {
    call SplitControl.start();
  }

event void SplitControl.startDone(error_t err) {
    if (err == SUCCESS) {
      call Timer.startPeriodic(TIMER_PERIOD_MILLI);
    }
    else {
      call SplitControl.start();
    }
  }

  event void SplitControl.stopDone(error_t err) {
  }
...
```

Amikor elindul a mote, azaz meghívódik a *Boot.booted()* event akkor bekapcsoljuk a rádiót. Ha a rádió sikeresen bekapcsolt, akkor elindítjuk a *Timer*-t periodikus üzemmódban, ha nem akkor újra megpróbáljuk elindítani a rádiót. A *SplitControl.stopDone()* event-et is deklarálnunk kell, még ha nem is használjuk, mert egy interface összes event-jét deklarálni kell. A *Timer.fired()* event-et egészítsük ki az üzenet elküldésével:

```
...
event void Timer.fired() {
  ...
  if (!busy) {
    BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)(call
AMSend.getPayload(&pkt, sizeof(BlinkToRadioMsg)));
    btrpkt->counter = counter;
    if (call AMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof(BlinkToRadioMsg))
== SUCCESS) {
      busy = TRUE;
    }
  }
}
...
```

Ha a *Timer.fired()* event generálódik, akkor ha a *busy=FALSE*, azaz az előző üzenet elküldése megtörtént, akkor az *AMSEnd.getPayload()* paranccsal egy mutatót kapunk a *message_t* struktúra *payload* területére. Ezt felhasználva berakjuk a *counter* értékét erre a területre, majd az *AMSend.send()* parancs segítségével továbbküldjük azt. Az *AM_BROADCAST_ADDR* azt jelenti, hogy broadcast-olva küldjük tovább az üzenetünket. Itt lehet egy érték is, és akkor az így meghatározott sorszámú mote felé történik a csomag továbbítása. Ha ez a parancs *SUCCESS*-el tért vissza, akkor a *busy=TRUE*, hisz az üzenetküldés elkezdődőt.

```
...
event void AMSend.sendDone(message_t* msg, error_t error) {
    busy = FALSE;
```

```
}
...
```
Végül az *AMSend.sendDone()* event-ben a *busy=FALSE*, hisz megtörtént az üzenet elküldése.

Hatodik lépésként módosítsuk a *BlinkToRadioC*-t is az alábbiaknak megfelelően:

```
...
implementation {
  ...
  components ActiveMessageC;
  components new AMSenderC(AM_BLINKTORADIO);
  ...
  App.AMPacket -> AMSenderC;
  App.AMSend -> AMSenderC;
  App.SplitControl -> ActiveMessageC;
}
```
Itt az *AM_BLINKTORADIO* az üzenetünk AM címét jelenti. Ez olyan, mint a számítógépek esetén a port. A mote azonosítója, pedig mint az IP cím. Az *AM_BLINKTORADIO*-t a header file-ban adhatjuk meg, a következő képen:

```
...
enum {
  AM_BLINKTORADIO = 6,
};
...
```
Hetedik lépésként egészítsük ki úgy a kódot, hogy a szomszédos mote-tól fogadja az üzeneteket, és a *counter* értékének alsó három bit-jét jelenítse meg a led-eken. Ehhez egészítsük ki a *BlinkToRadioP* modult a következő képen:

```
module BlinkToRadioC {
  ...
  uses interface Receive;
}

event message_t* Receive.receive(message_t* msg, void* payload, uint8_t
len)
{

    BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)payload;
    call Leds.set(btrpkt->counter);

    return msg;
}
```
A *Receive* interface biztosítja azt az event-et amely segítségével fogadni lehet az üzenetet, azaz a *Receive.receive()* event-et. Ha megjön az üzenet, akkor kivesszük a *counter* értékét és megjelenítjük az alsó három bit-jét a led-eken. Végül visszatérünk a kapott üzenettel. Ez nagyon fontos, mert a beérkező üzenetet a TinyOS egy buffer-be menti, és ha nem adjuk vissza a *Receive.receive()* event lekezelése után, akkor a következő beérkezett üzenetet nem tudja a TinyOS hova tenni. Természetesen a *BlinkToRadioC* konfigurációt is ki kell egészíteni:

```
implementation {
  ...
  components new AMReceiverC(AM_BLINKTORADIO);
  ...
  App.Receive -> AMReceiverC;
}
```

Itt is ugyanazt az AM azonosítót adjuk meg, tehát ez azt az üzenetet fogja fogadni, amit másik mote elküldött. Fordítás előtt mindkét file-ba include-oljuk be a *BlinkToRadio.h*-t Ezután fordítsuk le és programozzuk fel a kódot a mote-okra:

```
make install telosb
```
Annak érdekében, hogy lássuk milyen csomagok lettek elküldve, felprogramozok egy *BaseStation* mote-ot, és elindítok egy *Listener* alkalmazást, ami az összes a *BaseStation* által kapott üzentet nyers formában megjeleníti a konzolon. Az alkalmazás elindítása:
```
java net.tinyos.tools.Listen –comm serial@/dev/ttyUSB0:telosb
```
Ezek után a következő jelenik meg a képernyőn:

00 FF FF 00 01 02 22 06 01 18
00 FF FF 00 01 02 22 06 01 19
00 FF FF 00 01 02 22 06 01 1A

- Az első byte (00) azt jelenti, hogy megérkezett egy AM csomag.
- A második két byte a cél címét jelöli. Az (FF FF) jelenti azt hogy az üzenet broadcastolva volt.
- A következő két byte (00 01) a forrás címét jelöli. Ha nem adunk meg programozáskor értéket, akkor automatikusan az 1 értéket kapja a mote.
- A következő byte (02) a csomagban található adatmező hossza.
- A következő mező (22) a grup-ot jelöli.
- A következő mező (06) az AM címet jelöli.
- A következő két byte (01 0A) az elküldött számláló értéket jelöli.

Következő feladatként próbáljuk ki azt, hogy az egyik mote-ot felprogramozzuk 1-es mote-nak:
```
make install,1 telosb
```
Ennél a mote-nál módosítsuk úgy a *BlinkToRadioP*-ben az *AMSend.send()* parancsot, hogy ne *AM_BROADCAST_ADDR*-ot tartalmazzon, hanem egy 2-est. Ezzel azt érjük el, hogy az 1-es mote a 2-es mote-nak fog küldeni üzenetet. Hasonlóképpen a következő mote-ot programozzuk fel, mint 2-es mote:
```
make install,2 telosb
```
Itt az *AMSend.send()* parancsot pedig módosítsuk, hogy a 3-as mote-nak küldjön. Végül az utolsó mote-ot programozzunk fel mint 3-as mote:
```
make install,3 telosb
```
Itt pedig úgy módosítsuk az *AMSend.send()* parancsot, hogy az 1-es mote-nak küldjön. A kör bezárult, a *BaseStation*-t és a *Listen*-t futtatva a következő eredményt kapjuk:

00 00 03 00 02 02 22 06 00 01
00 00 02 00 01 02 22 06 00 01
00 00 01 00 03 02 22 06 00 01
00 00 03 00 02 02 22 06 00 02
00 00 02 00 01 02 22 06 00 02
00 00 01 00 03 02 22 06 00 02

Itt látható, hogy mit is csináltunk. A 2-es mote a 3-asnak küld, az 1-es mote a 2-esnek, és végül a 3-as mote az 1-esnek.