Mote-Mote rádiós kommunikáció

Egy olyan alkalmazást készítünk, amely bizonyos időközönként növeli egy számláló értékét, majd az új számláló értéket elküldi a rádión keresztül a szomszédos mote-nak. A fogadó mote pedig megjeleníti a számláló alsó 3 bitjének értékét a led-eken. Ez a folyamat visszafele is fog működni.

Első lépésként készítsünk egy *BlinkToRadio* mappát. Ebbe másoljuk, bele azt a *Blink* alkalmazást, amely egy *counter* értékét jeleníti meg a led-eken. Változtassuk meg a konfigurációs file nevét *BlinkToRadioC.nc*-re és a benne lévő konfiguráció nevét *BlinkToRadioC*-re. Az alkalmazásunk nevét is változassuk meg, ugyanebben a file-ban *BlinkToRadioP*-re. A modul file-t nevezzük át *BlinkToRadioP.nc*-re és a benne lévő modult is nevezzük át *BlinkToRadioP*-re. A *Makefile*-ban pedig a konfigurációs komponenst is nevezzük át *BlinkToRadioC*-re.

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;
```

A *header*, a *footer* és a *metadata* mezők tartalma az alkalmazott rádiós chip és protokoll függvényében változik. Számunkra a legfontosabb a *data* mező. A rádiós adatküldés során a *data* mezőbe fogjuk beírni a *counter* értékét, és így fogjuk elküldeni az üzenetet. A TOS_DATA_LENGTH adja meg a *data* terület méretét, ennek mérete alapból 28 byte, de növelhető egészen 144 byte-ig. Annak érdekében, hogy el tudjuk küldeni a *counter* értékét 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 csináljunk 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
```

Itt lényegében definiálunk egy struktúrát. Ebben nincs semmi különös. A különösség abban van, hogy mind a *struct*, mind pedig az *uint16_t*-t megelőzi 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. Miután definiáltuk az üzenet struktúránkat, tehát azt a formát, ahogy el szeretnénk küldeni az üzenetünket, nézzük, meg hogyan lehet az üzenetet elküldeni.
Ehhez nézzük meg, hogy milyen interface-ek is kellenek annak érdekében, hogy el tudjuk küldeni az üzenetet a rádión keresztül. Kell az *AMSend* és a *SplitControl* interface. Az

AMSend olyan parancsokat és eseményeket tartalmaz, amivel az elküldendő üzenetet be lehet tölteni a *message_t* struktúra *data* (*payload*) területére, illetve ami segítségével el lehet küldeni az üzenetet a rádión keresztül, és visszajelzést kaphatunk, ha az üzenetküldés megtörtént. A *SplitControl* interface pedig olyan parancsokat és eseményeket tartalmaz, amivel be vagy ki lehet kapcsolni egy eszközt (jelen esetben ez a rádió), és visszajelzést kapunk ennek megtörténtéről. A *BlinkToRadioP* modult ennek megfelelően egészítsük ki az *AMSend* és a *SplitControl* interface-el. Emellett include-oljuk be a *BlinkToRadio.h* header file-t.

```
#include „BlinkToRadio.h”

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

Deklaráljunk két változót, melyek szerepét a későbbiekben fogom bemutatni:

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

A következő lépésként annak érdekében, hogy használni tudjuk, a rádió-t be kell kapcsoljuk. Ezt a *command error_t SplitControl.start()* paranccsal tehetjük meg. A parancs visszaad egy *error_t* típusú változót, mely a bekapcsolási folyamat elindításának eredményéről tájékoztat (sikeres, nem sikeres, már el van indítva a rádió, stb...). A rádió bekapcsolásáról pedig az *event void SplitControl.startDone(error_t err)* esemény 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 Timer1.startPeriodic(TIMER_PERIOD_MILLI);
    }
    else {
      call SplitControl.start();
    }
  }

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

Itt tehát amikor elindul a mote, azaz meghívódik az *event void booted()* esemény akkor bekapcsoljuk a rádiót. Ha a rádió sikeresen bekapcsolt, akkor elindítjuk az időzítőt periodikus

üzemmódban (*command void Timer1.startPeriodic(uint32_t time)*) ha nem akkor újra megpróbáljuk elindítani a rádiót. Az *event void SplitControl.stopDone(error_t err)* eseményt is deklarálnunk kell, még ha nem is hívjuk meg hozzá tartozó *command error_t SplitControl.stop()* parancsot, hisz egy interface összes eseményét deklarálni kell, mert egyébként hibát kapunk. A következő lépés hogy az *event void Timer.fired()* részt egészítsük ki az üzenet elküldésével:

```
...
event void Timer1.fired() {
  ...
  if (!busy) {
    BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)(call
AMSend.getPayload(&pkt, sizeof(BlinkToRadioMsg)));
    btrpkt->counter = counter++;
    if (call AMSend.send(TOS_NODE_ID, &pkt,
sizeof(BlinkToRadioMsg)) == SUCCESS) {
      busy = TRUE;
    }
  }
}
...
```

Ha az *event void Timer1.fired()* event generálódik, akkor ha a busy=FALSE, azaz az előző üzenet elküldése megtörtént, akkor a *command void* AMSend.getPayload(message_t* msg, uint8_t len)* paranccsal egy mutatót kapunk a *message_t* struktúra *payload* területére. Ezt cast-oljuk *BlinkToRadioMsg* típusúvá. Erre azért van szükség, mert ezt felhasználva berakhatjuk a *counter* értékét erre a területre, majd megnöveljük annak értékét és a *command error_t AMSend.send(am_addr_t addr, message_t* msg, uint8_t len)* parancs segítségével továbbküldjük azt a rádión keresztül. A *TOS_NODE_ID* változó a fordításkor ki fog cserélődni arra az értékre, amit a *make* parancsban megadtunk. Ez lesz az adott mote azonosítója. Ez a mote csak ennek az azonosítójú mote-nak fog üzenetet továbbítani. Ha *AM_BROADCAST_ADDR*–t írunk be *addr*-ként akkor broadcast-olva küldjük tovább az üzenetünket, tehát minden mote megkapja azt. Az hogy most egy konkrét mote azonosítót adtunk meg, az a jelentősége, hogy mivel most több mote kommunikál egy időben, így mindegyik mind egyik másiktól megkapná az üzenetet, és nem lehetne követni a led-ek villogását. 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 *event void AMSend.sendDone(message_t msg, error_t error)* eventben a *busy=FALSE*, hisz megtörtént az üzenet elküldése. Most módosítsuk a *BlinkToRadioC*-t is az alábbiaknak megfelelően:

```
#include „BlinkToRadio.h”

implementation {
  ...
  components ActiveMessageC;
```

```
  components new AMSenderC(AM_BLINKTORADIOMSG);
  ...
  App.AMPacket -> AMSenderC;
  App.AMSend -> AMSenderC;
  App.SplitControl -> ActiveMessageC;
}
```

Itt az *AM_BLINKTORADIOMSG* az üzenetünk AM címét jelenti. Ez lényegében olyan, mint a számítógépek esetén a port. A mote azonosítója, pedig mint az IP cím. Az *AM_BLINKTORADIOMSG*-t a header file-ban adhatjuk meg, a következő képen:

```
...
enum {
  AM_BLINKTORADIOMSG = 6,
};
...
```

Következő lépésként egészítsük ki úgy a kódot, hogy a szomszédos mote-tól fogadja is az üzeneteket, és az üzenetbe elhelyezett *counter* értékének alsó három bit-jét jelenítse meg a led-eken. Ehhez egészítsük ki a *BlinkToRadioP* modult a következő képen:

```
module BlinkToRadioP {
  ...
  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 tartalmazza *event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len)* eseményt, amely segítségével fogadni lehet az üzenetet. Ha megjön az üzenet, akkor kivesszük a *counter* értékét és megjelenítjük az alsó három bit-jét a led-eken. Itt nem kell használni az *command void* AMSend.getpayload*() parancsot, hisz itt az *event message_t* Receive.receive()* esemény paraméterlistája már tartalmazza a fogadott üzenet *palyoad* területére mutató mutatót. Végül visszatérünk a kapott üzenettel. Ez nagyon fontos, mert a beérkező üzenetet a TinyOS egy pufferbe menti, és ha nem adjuk vissza az *event message_t* Receive.receive()* esemény lekezelése után a lefoglalt puffert, akkor egy idő után a következő beérkezett üzenetet nem tudja a rendszer 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 az interface azt az üzenetet fogja fogadni, amit a másik mote elküldött.

Ezután a *make install telosb, x* utasítással fordítsuk le és programozzuk fel a kódot a mote-okra. Az x jelenti a mote azonosítóját.

Annak érdekében, hogy lássuk milyen csomagok lettek elküldve, az egyik TelosB mote-ot felprogramozhatjuk *BaseStation*-ként. A *BaseStation* az összes rádiós üzenetet fogadja és továbbítja azokat a soros port-on a számítógép felé. El kell még indítani egy *Listener* alkalmazást, ami azt csinálja, hogy az összes a *BaseStation* által kapott üzentet nyers formában megjeleníti a konzolon. A *Listener* alkalmazás elindítása a következő képen történik. *java net.tinyos.tools.Listen –comm serial@/dev/ttyUSB0:telosb*

Ezek után a következő jelenik meg a képernyőn:

00 00 01 00 01 02 22 06 01 18
00 00 01 00 01 02 22 06 01 19
00 00 01 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.
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 group-ot jelöli.
A következő mező (06) az AM címet jelöli.
A következő két byte (01 0A) az elküldött számláló értéket jelöli.