Broadcast alkalmazás készítése

Feladatunk egy olyan alkalmazás készítése, amely a TelosB mote bekapcsolásakor broadcast-olva elküldi a mote saját belső azonosítóját (jelen alkalmazás esetén ez a TOS_NODE_ID), majd várja a szomszédos mote-ok által elküldött csomagok megérkezését. Ha megérkezik egy csomag, akkor az alkalmazás megvizsgálja, a csomagban lévő azonosítót. Ha ez az érték megegyezik a mote saját belső azonosítójával, akkor ezzel az értékkel felülírja azt és broadcast-olva továbbküldi a csomagot. Ha a két érték azonos, akkor a csomag eldobásra kerül. A belső változó aktuális értékét a mote-on található led-ek segítségével jelenítjük meg.

A feladat elkészítésének lépései:

1.lépés: Hozzunk létre egy teljesen üres mappát, és nevezzük el *Broadcast*-nak. Ebbe a mappába hozzunk létre egy *BroadcastC.nc* file-t. Ez a file fogja tartalmazni az alkalmazásunk konfigurációját. A konfiguráció tartalmazza azokat a komponenseket, illetve az összekapcsolásukat, melyek az alkalmazásunkhoz szükségesek. A *BroadcastC.nc* file-ba írjuk be az alábbi kódot:

```
configuration BroadcastC{
}
implementation
{
	components MainC;
	components LedsC;
	components new AMSenderC(0x1);
	components new AMReceiverC(0x1);
	components ActiveMessageC;
	components BroadcastP;

	BroadcastP.Boot->MainC;
	BroadcastP.Leds->LedsC;
	BroadcastP.AMSend->AMSenderC;
	BroadcastP.Receive->AMReceiverC;
	BroadcastP.SplitControl->ActiveMessageC;
}
```

A *MainC* komponens biztosítja a *Boot* interface-t, a *LedsC* komponens biztosítja a *Leds* interface-t, az *AMSenderC* generikus komponens biztosítja az *AMSend* interface-t, az *AMReceiverC* generikus komponens biztosítja a *Receive* interface-t, míg az *ActiveMessageC* komponens biztosítja a *SplitControl* interface-t. A *BroadcastP* az általunk készített alkalmazást tartalmazza. A konfigurációs file-ban, a felhasználni kívánt komponenseken kívül, még meg kell adni az egyes komponensek össze huzalozását is, tehát azt, hogy a *BroadcastP* komponens által felhasznált interface-eket, mely komponensek biztosítják.

2. lépés: Készítsük el a *BroadcastP* komponenst is, melyre az előbbiekben tárgyalt *BroadcastC* komponensben már hivatkoztunk is. Hozzuk létre a *BroadcastP.nc* file-t, mely majd az alkalmazásunkat (*BroadcastP*) fogja tartalmazni. A *BroadcastP.nc* file-ba írjuk be az alábbi kódot:

```
module BroadcastP{

	uses interface Boot;
	uses interface Leds;
	uses interface AMSend;
	uses interface Receive;
	uses interface SplitControl;
}
implementation{

	event void Boot.booted(){
	}

	event void SplitControl.startDone(error_t error){
	}

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

	event void AMSend.sendDone(message_t* msg, error_t error){
	}

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

Látható, hogy a *BroadcastP* komponens használja a *Boot*, *Leds*, *AMSend*, *Receive*, *SplitControl* interface-eket, melyeket a *BroadcastC.nc* file-ban össze is kötöttünk az őket biztosító komponensekkel. A felhasznált interface-ek a *(TOSROOT)/tos/interfaces* mappában találhatóak. Ha az itt található interface-eket megnézzük, akkor láthatjuk, hogy az interface-ek különböző *event*-eket, illetve *command*-okat tartalmaznak. Annak érdekében, hogy le tudjuk fordítani a kódunkat, a felhasznált interface-ek által biztosított összes event-et deklarálnunk kell. A *Boot* interface a *void booted()* event-et tartalmazza, mely azt jelzi, hogy a rendszer elindult. Az *AMSend* interface a *void sendDone(message_t* msg, error_t error)* event-et tartalmazza, mely azt jelzi, hogy az üzenetet elküldésre került. Az első paraméter az elküldött üzenetre mutató pointer, míg a második paraméter az üzenet küldésének eredményét adja vissza. A *Receive* interface a *message_t *receive(message_t* msg, void* payload, uint8_t len)* event-et tartalmazza, mely azt jelzi, hogy egy üzenet érkezett. Az első paraméter a kapott üzenetre mutató pointer, a második paraméter a kapott üzenet payload területére mutató pointer, míg a harmadik paraméter a payload terület hossza. A *SplitControl* interface pedig a *void startDone(error_t error)*, illetve a *void stopDone(error_t error)* event-eket tartalmazza, ahol a paraméter a bekapcsolás, illetve a kikapcsolás eredményét mutatja.

3. lépés: Hozzunk létre egy *Makefile* file-t, mely az előbbiekben elkészített alkalmazás fordításához szükséges. A *Makefile*-nak a következő két sort kell, tartalmazza:

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

Az első sor megadja a konfigurációs file nevét, mely az alkalmazás komponenst tartalmazza. A második sor, pedig meghívja a TinyOS build rendszerét, mely segítségével lefordítható az alkalmazásunk.

4. lépés: Fordítsuk le az alkalmazásunkat, az alábbi parancs segítségével:

```
make telosb
```

5. lépés: Mivel a feladatunk az, hogy a mote bekapcsolásakor elküldjünk broadcast-olva egy üzenetet, mely a saját belső változónk értékét tartalmazza, ezért a *void Boot.booted()* event-ben kapcsoljuk be a rádiót, és deklaráljuk a belső változónkat (*actualNumber*) az alábbi módon:

```
uint16_t actualNumber;

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

6. lépés: A rádió bekapcsolását a *void startDone(error_t error)* event jelzi. Ha a rádió bekapcsolódott, akkor definiáljuk a belső azonosítónkat (legyen az értéke a *TOS_NODE_ID*), másoljuk bele az elküldendő üzenetbe, majd küldjük el az üzenetet az alábbi módon:

```
message_t dataMsg;
bool busy=FALSE;

event void SplitControl.startDone(error_t error){
      nx_uint16_t *payload;
      actualNumber=TOS_NODE_ID;
      call Leds.set(actualNumber);
      payload=(nx_uint16_t*)call AMSend.getPayload(&dataMsg,2);
      *payload=actualNumber;
      if(call AMSend.send(AM_BROADCAST_ADDR,&dataMsg,2)==SUCCESS){
            busy=TRUE;
      }
}
```

Az *actualNumber* belső azonosító aktuális értékét a *Leds.set()* parancs segítségével tudjuk megjeleníteni a led-eken. Ha az üzenetküldés elkezdődik, azaz az *AMSend.send()* parancs *SUCCESS*-el tér vissza, akkor ezt egy *busy* flag beállításával jelezzük.

7. lépés: Módosítsuk az *AMSend.sendDone()* event-et úgy, hogyha megtörtént a csomag elküldése, akkor a *busy* flag-et állítsa *FALSE*-ra az alábbi módon:

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

8. lépés: Fordítsuk le az alkalmazásunkat és programozzuk fel a mote-ra, az alábbi módon:

```
make telosb install,4
```

Ha az alkalmazást sikeresen felprogramoztuk, akkor a led-eken a 4-es számnak megfelelő bináris minta fog megjelenni

9. lépés: A feladatnak megfelelően a szomszédos mote-ok felől érkező csomagokat is fogadnunk kell, és megnézni a megérkezett csomag tartalmát. Ha a csomagban lévő azonosító nem egyenlő a saját belső azonosítónkkal, akkor az új értéknek megfelelően módosítjuk a belső azonosítónk értéket, és broadcast-olva továbbküldjük a csomagot. Ha a két azonosító

megegyezik, akkor az üzenetet eldobjuk. Az aktuális azonosító értékét szintén jelenítsük meg a led-eken. A feladat elvégzéséhez az alábbi módon kell kiegészíteni a *Receive.receive()* event-et:

```
        message_t *freeMsg;

        event message_t* Receive.receive(message_t* msg, void* payload,
                                         uint8_t len){
        if(busy)
              return msg;
        if(*(nx_uint16_t*)payload==actualNumber)
              return msg;
        if(call AMSend.send(AM_BROADCAST_ADDR,msg,len)!=SUCCESS)
              return msg;

        busy=TRUE;
        actualNumber=*(nx_uint16_t*)payload;
        call Leds.set(actualNumber);
        return freeMsg;

}
```

Elsőként megvizsgáljuk, hogy éppen nem küldünk-e csomagot, azaz megvizsgáljuk hogy a *busy* flag *TRUE*-e. Ha igen, akkor a csomagot eldobjuk, és a lefoglalt *message_t* struktúrát visszaadjuk a rádió stack-nek, annak érdekében, hogy újra fel tudja használni azt az érkező csomagok fogadásához. Következő lépésben megvizsgáljuk, hogy a csomagban lévő azonosító egyezik-e a saját belső azonosítónkkal, ha igen, akkor szintén eldobjuk a csomagot és visszaadjuk a *message_t* struktúrát. Harmadik lépésben, pedig meghívjuk az *AMSend.send() command*-ot, mely segítségével a beérkezett üzenetet broadcast-olva továbbküldjük. Ha ez a *command* nem *SUCCESS*-el tér vissza, azaz az üzenet elküldése sikertelen, akkor szintén visszaadjuk a *message_t* struktúrát a rádió stack-nek, hisz tovább már nincs rá szükségünk. Ha viszont az *AMSend.send() command SUCCESS*-el tér vissza, ami azt jelenti, hogy elkezdődött a rádiós adatküldés, akkor a *busy* flag-et *TRUE*-ra állítjuk, és mivel a saját belső azonosítónk nem egyezik meg a kapott azonosítóval, ezért ezt az új értéknek megfelelően módosítjuk, és az aktuális értékét megjelenítjük a led-eken a *Leds.set()* parancs segítségével. Mivel a beérkező üzenetet tartalmazó *message_t* struktúrát átadtuk a rádió stack-nek elküldésre, ezért nem adhatjuk vissza, hogy az újonnan érkező üzeneteket ebbe rakja a radio stack, ezért egy üres *message_t* struktúrát fogunk visszaadni, melyet az alábbi módon inicializálunk.

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

A *freeMsg*-t egyenlővé tesszük az *AMSend.sendDone()* event által visszaadott *message_t* struktúrával, mely az első adatküldéskor a *dataMsg*. A további adat küldésekkor, pedig a továbbküldésre lefoglalt *message_t* struktúrákra fog mutatni a *freeMsg*.

10. lépés: Fordítsuk le az alkalmazásunkat, és töltsük fel a mote-okra. Helyes működés esetén mindig az utolsóként bekapcsolt mote belső azonosítóját veszi fel a többi mote, és jeleníti meg bináris formában a led-eken.

Az elkészített alkalmazásunk azonban magában hordozza a hiba lehetőségét, és végtelen ciklusban folyó, ping-pong jelenség alakulhat ki. Vizsgáljuk meg az alábbi eseménysorozatot:

- 1-es mote elküldi az 1-es értéket
- 2-es mote elküldi a 2-es értéket
- 1-es mote-nál *sendDone()* event generálódik
- 2-es mote-nál *sendDone()* event generálódik
- 2-es mote megkapja az 1-es értéket az 1-es mote-tól és felülírja a saját belső változóját
- 1-es mote megkapja a 2-es értéket a 2-es mote-tól és felülírja a saját belső változóját
- 2-es mote elküldi az 1-es értéket
- 1-es mote elküldi a 2-es értéket
- mivel az üzenetváltás végén a két mote-nak különböző belső változó értékei vannak az egész folyamat kezdődik előröl.

Látható, hogy az üzenetküldés sohasem fog megállni, mert a két mote-on a belső változó értéke mindig különböző lesz. A *Receive.receive()* event-ben ha a *busy* flag *TRUE*, akkor eldobjuk az üzenetet. Be lehetne vezetni egy plusz állapotot, melyben jelzzük, hogy a küldés alatt egy új üzenet érkezett, azonban ez sem oldja meg tökéletesen a problémát. Nézzük meg a következő eseménysort:

- 1-es mote elküldi az 1-es értéket
- 2-es mote elküldi a 2-es értéket
- 2-es mote megkapja az 1-es értéket az 1-es mote-tól
- 1-es mote megkapja a 2-es értéket a 2-es mote-tól
- 1-es mote-nál meghívódik a *sendDone()* event
- 1-es mote elküldi a 2-es értéket
- 2-es mote-nál meghívódik a *sendDone()* event
- 2-es mote elküldi az 1-es értéket
- kezdődik az egész folyamat előröl

A probléma kiküszöbölésére definiálnak egy kitüntetett mote-ot, amely bizonyos időközönként küld egy broadcast üzenetet, melyben megadnak egy sorszámot, amelyet ez a mote növel minden broadcast üzenetküldéskor. A többi mote pedig csak akkor küldi tovább broadcast-olva ezt az üzenetet, ha a sorszám nagyobb, mint az előzőleg elküldött csomagban lévő sorszám. A nagyobb eldöntés is problémákba ütközhet, hisz akármilyen nagy bitszélességű változót használunk előbb, vagy utóbb túlcsordulhat. Ezért az alábbi módon dönthetjük el két értékről, hogy melyik a nagyobb:

```
uint8_t(a-b)
```

Ha ez az érték nagyobb, mint 0 akkor az a nagyobb, mint a b.