TinyOS operációs rendszer portolása *mitmót*ra

Siroki László V. inf.

2006

Konzulens: dr. Tóth Csaba, Méréstechnika és Információs Rendszerek Tanszék

Tartalomjegyzék

1. Bevezetés

A mikroelektronika fejődésével megnyílt a lehetőség arra, hogy nagyon kis méretű, kis fogyasztású, hosszú élettartamú, érzékelésre, beavatkozásra és kommunikációra képes eszközöket gyártsanak, így kialakult egy új kutatási és ipari ágazat, ami a vezeték nélküli szenzorhálózatokkal foglalkozik. A BME Méréstechnika és Információs Rendszerek Tanszékén néhány évvel ezelőtt fejlesztették ki a moduláris „mitmót" rendszert, ami egy ilyen szenzorhálózati platform. Ekkor vezették be, hogy a Beágyazott Információs Rendszerek villamosmérnöki főszakirány hallgatói kézhez kapnak egy-egy példányt ebből, ennek otthoni fejlesztéséhez szükséges szoftverekkel együtt. Ezzel szeretnék elérni, hogy valamennyi főszakirányos hallgatónak készségszintű hardver- és szoftverismeretei legyenek, mire végez.

A tanszék munkájába 2005 őszén csatlakoztam be, önálló laboratóriumi munka keretében. Feladatom az volt, hogy ismerkedjek meg az általunk elérhető egyszerűbb beágyazott operációs rendszerekkel, és válasszam ki közülük azt, amelyet a legjobbnak találok a nyolcbites mitmót vezérlőkártya számára. A programozás eddig *assembly* és *C* nyelven történt, viszont ilyen környezetben a hallgatók egy-két félév alatt csak odáig tudnak eljutni, hogy megismerik a hardver lehetőségeit, a programozás mikéntjét és buktatóit, készítenek valamilyen saját kiegészítő modult hozzá, és megírják a teszteléséhez szükséges programot. A különböző hallgatók által írt programok egyesítése egy alkalmazásban nehézkes, és bonyolultabb, valódi szenzorhálózati feladatok megoldásához ez a környezet kevés, ahhoz több támogatás kell, ezért kellett keresni valamilyen operációs rendszert.

A 2. fejezetben bemutatom, hogy milyen operációs rendszerek jöttek szóba. A 3. fejezet arról szól, hogyan ismerkedtem meg a TinyOS-szel, mielőtt a portolását elkezdtem, mert először meg kellett tanulni felhasználói szinten kezelni, mielőtt nekifogtam a kitűzött célnak. A 4. fejezetben azt írom le, hogy milyen teendőket kellett elvégezni ahhoz, hogy az új platformon működjön az operációs rendszer. Ez egy eléggé aluldokumentált része a TinyOS-nek, így magamtól kellett felderíteni; azt írom le, amire jutottam. Az 5. fejezetben a mitmóthoz készült egyszerű perifériakártya támogatását ismertetem. A 6. fejezet a rádiós kártya támogatásáról szól, ez volt a portolás legtöbb munkát igénylő része. A 7. fejezetben néhány működő alkalmazást írok le, melyeknek egy része a TinyOS-szel együtt jár, és minden platformon működik, más része a mitmót perifériáinak tesztelésére készült, valamint olyanok is akadnak, melyek saját fejlesztésűek, és konkrét gyakorlati feladatokra adnak megoldást. A 8. fejezetben ismertetem a jövőbeli terveimet a téma folytatását illetően, a 9. fejezet pedig az irodalmi hivatkozásokat tartalmazza.

2. Beágyazott operációs rendszerek

Sokféle beágyazott operációs rendszer létezik, vannak közöttük ingyenesek és fizetősek, kisebb és nagyobb teljesítményű processzorra tervezettek, szegényesebb és gazdagabb funkcionalitásúak. A feladatomban egy 8 bites Atmel AVR ATMega128L mikrokontrollert [1] tartalmazó processzorkártya volt a rendszer alapja (a mitmótoknak [2] egyébként létezik egy másik, ARM processzoros változata is), ami erősen meghatározza a programozási környezet és paradigma kiválasztását. Több lehetőség is nyílik az ilyen kisteljesítményű rendszerek programozására, a leginkább elterjedteket ismertetem a következő alfejezetekben.

2.1 μC/OS

Ez egy mikrokontrollerek számára kifejlesztett valós idejű (*real time*) beágyazott operációs rendszer, jelenlegi verziója a μC/OS-II [3]. Ez a rendszer úgy épül fel, mint általában egy operációs rendszer: van egy magja, amely tartalmazza a preemptív multitaszkingot lehetővé tevő ütemezőt, emellett különféle rendszerszolgáltatásokat nyújt, például a szemafor, postaláda, üzenetsor, időzítési szolgáltatások. Megszakítás-kezelési lehetőséggel is rendelkezik, a megszakítások egymásba ágyazhatók. Programot írni hozzá hagyományosan C nyelven lehet, az OS függvénykönyvtárainak felhasználásával.

Előnye ennek a rendszernek, hogy hordozható (a processzorfüggő részek külön vannak implementálva) és skálázható, gyors, nagyon megbízható. A skálázhatóság itt azt jelenti, hogy konfigurálni lehet, hogy milyen szolgáltatások milyen paraméterrel legyenek elérhetőek. Például beállítható az egyszerre futó taszkok maximális száma, hogy akarunk-e használni üzenetsorokat, és azon belül is milyen funkciókra van vagy nincsen szükségünk. Az operációs rendszer forráskód szinten áll rendelkezésre, így ezt a konfigurálhatóságot feltételesen definiált kódblokkok használatával érték el. Amelyik szolgáltatást nem használjuk, az így nem fog belekerülni a programba, ez a helytakarékosság miatt szükséges is. Előny még az is, hogy determinisztikus és konstans a legtöbb operációs rendszerhívás futási ideje, így kemény valós idejű (*hard real time*) követelményeknek is meg tud felelni.

Ez az operációs rendszer tényleg szép, jó, általános, sokoldalú, megbízható, de nem kifejezetten szenzorhálózatokhoz találták ki, és hátránya, hogy hiányzik belőle az objektum-orientáltság, ami átláthatóságot, újrafelhasználhatóságot és könnyű továbbfejlesztést és bővíthetőséget hozna magával.

2.2 TinyOS

A TinyOS operációs rendszert [4] a kaliforniai Berkeley egyetemen fejlesztették ki, speciálisan a szenzorhálózati alkalmazások igényeihez igazítva. Szorosan kötődik hozzá a nesC nyelv, amelyben a TinyOS komponenseket megvalósították, és amiben programokat lehet írni az operációs rendszer alá. A nesC a C nyelv egy komponens-orientált és eseményvezérelt szemléletű, bővített változata, ezzel követi a modern szoftverfejlesztési paradigmákat. A TinyOS/nesC rendszerről az [5], [6] és [7] irodalmakból tájékozódtam.

A nesC nyelvű programok komponensekből állnak, amelyek kapcsolatban vannak egymással. A komponensek megvalósítnak valamilyen feladatot, a többi komponens felé szolgáltatásokat nyújthatnak, és a feladatuk végrehajtásához maguk is igényelhetnek más komponenseket. Ez az architektúra kínálja a funkcionális szétválasztás és így a cserélhetőség lehetőségét. A komponensek parancsok és események segítségével kommunikálnak egymással. A *parancs* (*command*) és az *esemény* (*event*) olyan függvények, amelyeket az egyik komponens megvalósít, és ezt más komponens meghívhatja. A különbség ott jelentkezik, hogy ha ezeket egy *interfész*be fogjuk össze, akkor különböző irányú kommunikációt tesznek lehetővé. Bár van lehetőség arra, hogy egy komponens különálló parancsokat vagy eseményeket nyújtson vagy használjon, az objektum-orientált szemlélet jegyében az esetek túlnyomó részében interfészeket *nyújt*anak (*provides* kulcsszó) vagy *használ*nak (*uses* kulcsszó). A komponenseket ezek mentén kapcsolhatjuk össze. Ha egy komponens nyújt valamilyen interfészt, akkor az megvalósítja az interfészben lévő parancsokat, valamint kiválthatja (*signal* kulcsszó) az abban szereplő eseményeket. Ha egy másik komponens használ valamilyen interfészt, akkor az azt jelenti, hogy meghívhatja (*call* kulcsszó) a benne lévő parancsokat, cserébe viszont kezelnie kell az interfészben leírt eseményeket.

A komponensek tehát nyújtanak és használnak interfészeket, de egy felsőbb szinten meg kell mondani, hogy hogyan kapcsolódjanak egymáshoz. Ez az összeköttetés az úgynevezett *konfigurációk*ban (*configuration*) történik meg. A konfigurációk maguk is komponensek, tehát úgy néznek ki, hogy maguk is nyújthatnak és használhatnak interfészeket, de felépítésük olyan, hogy belül nem imperatív programkódot, hanem más komponenseket tartalmaznak, amiket „összehuzaloznak” az összeillő interfészeik mentén (egy interfészt használó komponenst az ugyanolyan interfészt nyújtóval lehet összekötni). Ezek úgy nyújthatnak és használhatnak interfészeket, hogy valamelyik belső komponens interfészét

vezetik ki, tehát csak delegálják a hívásokat, így egy hierarchikus rendszer kialakítását teszik lehetővé.

A komponensek másik típusa az úgynevezett *modul* (*module*). A moduloknak már egyszerű a belső szerkezete, ezek vannak a hierarchikus felépítés alján, mert a modulok tartalmazzák a parancsok és eseménykezelők konkrét implementációit. A függvények megvalósítása már majdnem teljesen C nyelvű, eltekintve a *call*, *signal* és *post* kulcsszavaktól, valamint néhány megkötéstől, mint például nem használhatunk dinamikus memória-allokációt és függvénymutatókat. A modulokban a parancsok és események megvalósításán kívül lehetnek még egyszerű *C* függvények és *taszk*ok is. A taszk egy olyan függvény, amit a *task* kulcsszóval deklarálunk, visszatérési értéke és argumentuma nincsen. A TinyOS ütemezője ilyen taszkokat futtat, egymás után (FIFO ütemezéssel), nem preemptív módon. A *post* kulcsszóval indíthatunk el egy új taszkot, ami le fog futni valamikor, ha majd rá kerül a sor.

Itt érdemes néhány szót ejteni az ütemezésről. Mivel az ütemező prioritás nélküli és nem preemptív, kemény valós idejű rendszerek készítésére nem a legalkalmasabb. A TinyOS fejlesztői úgy indokolják mégis ezt a döntést, hogy a szenzorhálózatokban a rádió bizonytalansága miatt amúgy sem várhatunk el minden téren szigorú követelményeket az időzítésre, a preemptív ütemezőnél jelentkező szinkronizációs problémákat viszont ez a megoldás kiküszöböli, mert minden taszk egymástól teljesen függetlenül fut, nem szakítják meg egymást. Van azonban mégis egy kis konkurencia a rendszerben, mert a megszakítások a taszkok futása közben is érkezhetnek, meghívhatnak parancsokat, vagy kiválthatnak eseményeket. Ahhoz, hogy a helyes működés ellenőrizhető legyen, az ilyen, megszakításból elérhető parancsokat és eseményeket az *async* kulcsszóval kell megjelölni, és ezeket *aszinkron kód*nak (*AC*) nevezzük. A többi függvény, tehát azok, amelyek csak taszkokból vannak meghívva, a *szinkron kód*okhoz (*SC*) tartoznak. Azért tudjuk ilyen egyértelműen eldönteni, hogy mi tartozik a szinkron és aszinkron kódokhoz, mert a hívási gráf (az, hogy melyik függvény melyiket hívhatja) már fordítási időben ismert, a függvénymutatók használatának tilalma miatt. A szinkron kódok között nem lehet ütközés, teljesen függetlenül futnak, egymás után. Az aszinkron kódok viszont megszakíthatják a szinkronokat vagy egymást, így szükség lehet arra, hogy olyan tartományokat hozzunk létre, amit nem szakíthat meg más. Ilyen eset például, ha több byte méretű változót másolunk, és egy másik szálból pedig olvasni akarjuk. Ekkor előfordulhatna, hogy egy félig átírt változót olvasnánk ki. Egy másik tipikus eset, ha egy állapotváltozót kiolvasunk, az értékének megfelelően végrehajtunk valamit, és új értéket adunk neki. Egy ilyen állapotváltozót több helyen is használhatunk,

például egymástól független események kezelésekor, és nem lenne szerencsés, ha az egyik helyen lefutna a kiolvasás, majd a másik szál megszakítaná, kiolvasná és megváltoztatná az értéket, és ezután folytatódna a futás az első szálban. Valószínűleg nem a tervezett működést kapjuk ebben a helyzetben. A probléma megoldására az atomi módon lefutó blokkok szolgálnak, amik minden esetben megszakítás nélkül futnak le. Ilyeneket az *atomic* kulcsszóval hozhatunk létre, és a TinyOS jelenlegi implementációjában a megszakítások időleges tiltásával valósítják meg.

Egy kicsit az ütemezéshez kapcsolódik még az energiatakarékosság kérdése. Azzal, hogy bizonyos perifériákat kikapcsolunk és a processzort a lehető legalacsonyabb fogyasztású állapotba helyezzük, ahol még működik, rengeteg energiát takaríthatunk meg. Az alvási módok használata a TinyOS-ben úgy van megoldva, hogy ha az ütemező azt találja, hogy nincsen több futtatandó taszk, akkor megvizsgálja, milyen perifériák működnek még, hogy meghatározza a lehető legalacsonyabb energiafogyasztású alvási szintet, ahonnan még a perifériák biztosan fel tudják ébreszteni, és elalszik. Lehet, hogy a rádió például be van kapcsolv, és arra fog felébredni, ha csomag érkezik, de az is gyakori, hogy a rádiót is kikapcsoljuk az alacsony fogyasztás érdekében, de beállítunk egy időzítőt, ami majd felébreszti a processzort. A lényeg tehát, hogy a processzor nem jár üresjáratban: ha éppen nem kell csinálni semmit, akkor alszik. Emiatt fontos, hogy tényleg mindent eseményvezérelt szemlélettel írjunk meg, és ne legyen a programokban üresen várakozó ciklus.

A TinyOS előre megírt komponensekből áll, ezek helyettesítik a hagyományos operációs rendszerek rendszerhívásait. Ezek elfedik a hardverek közötti különbséget, mert minden platformra külön meg vannak írva a hardverfüggő részek, így a programozást már csak egy magasabb, absztrakt szinten kell elvégezni. Emellett azzal az előnnyel is jár ez a megoldás, hogy már az operációs rendszer is komponens-orientált szemléletű, így jól illeszthető hozzá ilyen program. A TinyOS/nesC programozás gyakorlatban úgy folyik, hogy el kell készíteni a saját interfészeket és komponenseket nesC nyelven, ezek között kell lennie egy legfelső szintű konfigurációnak. Fordításkor egy előfordító egy C nyelvű kódot készít a NesC forrásokból, majd hagyományos C fordítóval készül el a gépi kód, amit be lehet programozni a mote-okba. A TinyOS helytakarékos, mert csak azokat a programrészleteket fordítja bele az elkészülő kódba, amelyeket a legfelső szintű konfigurációban közvetlenül vagy közvetve használjuk. A μC/OS-szel ellentétben itt nem külön konfigurációs fájlban kell jelezni, hogy mire van szükségünk, hanem egyszerűen használni kell (persze ehhez valamelyik konfigurációban le kell írni a nevét), és a fordító rájön.

3. Előkészületek – ismerkedés a TinyOS-szel

Ahhoz, hogy belekezdhessek a portolásba, először alaposabban meg kellett ismernem a rendszert. Kaptam kölcsön 2 db Berkeley MICAz mote-ot [8, 10], programozó készüléket és szoftvertelepítő CD-t, amivel nekiállhattam a munkámnak. A telepítés a dokumentáció alapján rendben ment, a cygwin rendszert, a tinyos-t és a kiegészítő csomagokat sikerült feltennem, csak a programozó (MIB600, Ethernetes változat [9, 10]) működésre bírása volt kicsit nehézkesebb, mert ez kevésbé volt jól dokumentált. A programozónak először egy konfigurációs programmal be kell állítani egy fix IP-címet abból a tartományból, amilyen hálózatra csatlakoztatva van, majd a webes konfigurációs felületén beállítani a benne található 2 UART perifériát megfelelő bitsebességekre, mivel a programozó készüléken egy Atmel mikrokontroller található, ami a soros portján keresztül kommunikál a PC-vel, és képes programozni a rá csatlakoztatott mote-ot. Az Ethernet felület csak a közvetlen soros kapcsolat kiváltására szolgál. A gond ezzel csak az, hogy az információkat több különböző helyről kellett összeszedni hozzá, de a lényeg, hogy végül működött.

Az első alkalmazás, amit kipróbáltam, a Blink volt, ami nem tesz mást, mint a mote-on található piros LED-et villogtatja 0,5 Hz-es frekvenciával. Miután ez működött, a rádió tesztje következett: Az egyik mote folyamatosan számolt és a számláló állását mindig elküldte rádión (ez a CntToLedsAndRfm alkalmazás), a másik pedig kijelezte a LED-jein a kapott érték utolsó 3 bitjét (ez az RfmToLeds alkalmazás). Ez is működött. Átnéztem a TinyOS/nesC programozásról szóló dokumentumokat [5, 6, 7], és első lépésként egy olyan egyszerű programot tűztem ki célnak, ami a rádión kapott számnál eggyel nagyobbat jelez ki. Első gondolatra ez egyszerűnek tűnik, de az eredeti alkalmazás csak abból állt, hogy tartalmazott egy RfmToInt komponenst, ami közvetlenül rá volt kötve az IntToLeds komponensre. A feladat megvalósításához tehát új modult kellett létrehozni, és bekötni az RfmToInt és az IntToLeds közé, de nem ez az egyetlen mód. A feladat egyszerűnek tűnik, de megfelelő arra, hogy a megtanult fogalmak tisztázódjanak bennünk, és végül utólag már tényleg egyszerű lesz az egész. Furcsának tűnhet, hogy egy ilyen egyszerű módosításhoz új modult kell létrehozni, amikor éppen az volt a cél az operációs rendszer használatával, hogy megkönnyítse a fejlesztést. Ez még nem ellentmondás, mert egyrészt meg lehetett volna oldani a feladatot úgy is, hogy az egyik modulban hozzáadunk egyet az egyik függvényhívás paraméteréhez, de mivel az egy közösen használt modul, le kellett volna másolni az alkalmazás könyvtárába, ami nem szép megoldás. A könnyű fejlesztést nem az biztosítja,

hogy nagyon rövid kódot kell írni, hanem az, hogy átlátható és könnyen bővíthető a rendszer, a jól megírt részek pedig újrafelhasználhatóak.

A rádiós kommunikáció egyszerű kezelése mellett az időzítők használata volt számomra az a szolgáltatás, amit a leghasznosabbnak találtam. Ezt a TimerC komponens nyújtja egy paraméteres interfészen keresztül, amiről többet a TinyOS dokumentációjából [7] tudhat meg az érdeklődő olvasó, mert nem célom, hogy teljesen ismertessem a dolgozatomban a nesC programnyelv összes lehetőségét. A TinyOS tartalmaz még szabványos interfészeket a szenzorok kezelésére, mert a legtöbb meglévő szenzort kezelő komponens az ADC interfészt nyújtja a szenzor által mért érték lekérdezésére: ez az interfész egy egyszeri és egy periodikus lekérdezést kezdeményező parancsot tartalmaz, valamint egy eseményt, amivel visszajelez, ha a mintavétel megtörtént és rendelkezésre áll a mérési adat, amit az esemény 16 bites paraméterében ad vissza. A 16 bites felbontás a gyakorlatban előforduló mérési felbontások és pontosságok mellett általában elegendő, a mikrokontrollerbe beépített A/D konverter például csak 10 bites.

4. A portolás menete

Ahhoz, hogy a portolásba belekezdhessek, nem csak felhasználói szinten kellett megismerkednem az operációs rendszerrel, hanem ennél mélyebben, valamint szükséges volt a hardverek alaposabb tanulmányozása is, hogy a MICAz mote-ról áttérhessek a mitmótra.

Először tehát feltérképeztem, hogy hogyan működik a fordítás (4.1 fejezet), mi a funkciója a könyvtárszerkezet különböző pontjain található állományoknak (4.2 fejezet), hogyan van megoldva a különböző platformok kezelése (4.3 fejezet). Ezeket tényleg fel kellett térképezni, mert a hivatalos dokumentációkban nem találtam elég ezekre vonatkozó információt. A Google kereső és a könyvtárszerkezet állományainak tartalmában való szövegkeresés segített a célom elérésében. Ezen kívül olyan módszert is alkalmaztam, hogy egy-egy fájl tartalmát ideiglenesen elrontottam szintaktikailag, így fordításkor kiderült, hogy éppen használja-e azt a fordító. A 4.4 fejezetben összehasonlítok néhány meglévő, hasonló mote típust a mitmóttal, majd a 4.5 fejezetben írom le, hogy milyen lényeges lépésekből állt a portolás.

4.1 A fordítási rendszer feltérképezése

A fordításhoz a TinyOS a make segédprogramot használja, minden alkalmazás könyvtárában kell lennie egy Makefile-nak. Ez az /opt/tinyos-1.x/apps könyvtár alkönyvtáraiban található alkalmazásoknál általában a következő felépítésű:

```
COMPONENT=CntToLedsAndRfm
PFLAGS += -I%T/lib/Counters
include ../Makerules
```

Az első sor megadja a legfelső szintű konfiguráció nevét.

A PFLAGS+=… utasítással a fordítónak adhatunk meg további paramétereket, tipikusan könyvtárakat a –I opcióval, ahol szeretnénk még, hogy megtalálja a forrásfájlokat.

Az utolsó, include-os sor mondja meg, hogy vegye figyelembe az apps könyvtárban lévő Makerules fájlban leírtakat is, ami tehát olyan dolgokat tartalmaz, ami minden alkalmazásra közösen vonatkozik, és tartalmazza a fordításhoz szükséges szabályokat. Az előbb említett módszert már felváltotta egy újabb, amit például az /opt/tinyos-1.x/contrib/xbow/apps könyvtárban található alkalmazások is használnak: az utolsó sorban

```
include $(TOSROOT)/tools/make/Makerules
```

áll. Ez az /opt/tinyos-1.x/tools/make/Makerules fájlra hivatkozik, ami tartalmaz egy olyan szkriptet, ami a fellelhető .target, .rules és .extra kiterjesztésű konfigurációs fájlok alapján összeállítja, mit hogyan kell fordítani. A .target állományokkal lehet új fordítási célt (platformot) definiálni, ennek a neve szerepel mindig a make parancs után közvetlenül, majd utána lehet extrákkal kiegészíteni, mint például az install és reinstall, amik a fordítással egybekötött programletöltést és a sima programletöltést végzik az eszközre. Ezek után megadható még opcionálisan a programozó készülék típusa és annak csatlakozási helye is. Ezeknek a további paramétereknek lehet jelentést adni a .extra fájlok létrehozásával. Ezek a konfiguráció állományok a /opt/tinyos-1.x/tools/make könyvtárban találhatók, azonban ennek vannak alkönyvtárai is, például az „avr”, „msp”, „pc” könyvtárak, melyek a platform legfontosabb elemének, a processzortípusoknak felelnek meg, tehát ez határozza meg, milyen fordítóprogramot kell használni. Ezeknek az alkönyvtáraknak tartalmazniuk kell a nevükkel megegyező nevű, .rules kiterjesztésű állományt, az őket használó .target fájlokban pedig szerepel a

```
$(call TOSMake_include_platform, <könyvtár neve>)
```

sor. A fordítási cél (target) tehát meghatározza a processzor platformot is, így a megfelelő .rules, .target és a még megadott .extra fájlok alapján az alkalmazás könyvtárában lévő Makefile és az esetlegesen még beillesztett, összes alkalmazásra kiható Makerules vagy hasonló fájl alapján már tényleg összeáll egy olyan Makefile, ami tartalmazza a fordításhoz szükséges összes információt.

4.2 A könyvtárszerkezet

A TinyOS rendszer Linux/UNIX vagy cygwin környezetben használható, telepítés után kialakul egy könyvtárszerkezet, amiben az operációs rendszer forrásállományai és a fordításhoz szükséges segédprogramok valamint konfigurációs állományok találhatók. A fontosabb könyvtárak szerepét ismerteti a 4.1 táblázat.

/opt/tinyos-1.x	A TinyOS gyökérkönyvtára.
↳ apps	Az alkalmazások találhatóak itt.
↳ Blink	LED-villogtató alkalmazás.
↳ CntToLedsAndRfm	Egyszerű rádiós adóalkalmazás.
↳ RfmToLeds	Egyszerű rádiós vevőalkalmazás.
↳ doc	A dokumentációk könyvtára.
↳ tools	Segédprogramok és konfigurációs állományaik helye.
↳ make	Itt találhatóak a .target és .extra fájlok és a Makerules.
↳ avr	AVR-specifikus dolgok, főként programozók .extrá-i.
↳ java	Java nyelvű segédprogramok.
↳ tos	
↳ interfaces	Alapvető interfészek.
↳ lib	Segédkönyvtárak helye.
↳ CC2420Radio	A micaz rádiójának kezelésére szolgál.
↳ Counters	Egyszerű demonstrációra szolgáló komponensek.
↳ platform	A platform-specifikus komponensek helye.
↳ avrmote	AVR alapú mote-ok közös komponensei.
↳ mica	
↳ mica128	
↳ mica2	
↳ micaz	
↳ mitmot	Az új platform helye.
↳ sensorboards	Szenzorkártyák komponenseit tartalmazó könyvtárak.
↳ system	Alapvető rendszerkomponensek
↳ types	Néhány .h fájl, ami típusdefiníciókat tartalmaz
↳ contrib	Külső társaságok kiegészítései
↳ xbow	A CrossBow cég (a Berkeley mote-ok gyártója) kiegészítései a TinyOS-hez. Itt hasonló könyvtárszerkezet található, mint a főkönyvtárban, az ezekben található fájlok felülbírálják az eredetieket.
↳ apps	
↳ tos	
↳ interfaces	
↳ lib	
↳ platform	
↳ system	
↳ bme	Az általam létrehozott új ág, itt találhatóak a platform-specifikus komponensek megvalósításai és az új alkalmazások.
↳ apps	
↳ tos	
↳ lib	
↳ IOModule	
↳ IA4420Radio	
↳ system	

4.1 táblázat

4.3 A különböző hardver platformok kezelése

A különböző platformok támogatására az idők folyamán többféle megoldást találtak ki, ezek közül többnek a nyomait is megtaláltam a könyvtárszerkezet vizsgálata közben. A jelenleg használt módszer az, amit a 4.1 fejezetben leírtam: a .target, .extra és .rules konfigurációs állományok és az /opt/tinyos-1.x/tools/make/Makerules használata. Található azonban egy Makerules fájl az /opt/tinyos-1.x/apps könyvtárban is, ami csak néhány fix platformot támogat, ami bele van írva. Ez egy régebbi megoldás volt. A Crossbow cég az újabb megoldást használja, de kiegészíti a saját apps könyvtárában lévő MakeXbowlocal állománnyal, amiben konfigurációs beállításokat állíthatunk be úgy, hogy az minden alkalmazás fordításakor egységes legyen.

Én a saját megoldás kialakításakor figyelembe vettem, hogy az új, bővíthető rendszert alkalmazzam, lehetőséget teremtsek az alkalmazások közötti közös paraméterállításra, és a megoldás illeszkedjen ahhoz az Eclipse kiegészítőhöz is, amit a TinyOS fejlesztéshez legjobbnak találtam [11, 12]. Ezen szempontok figyelembevételével létrehoztam a tinyos-1.x/contrib/bme/apps könyvtárban egy Makerules fájlt, ami a közös beállításokat tartalmazza, és a végén hivatkozik az /opt/tinyos-1.x/tools/make/Makerules fájlra. Így a programokhoz tartozó Makefile-okban az

```
include ../Makerules
```

sort kell szerepeltetni.

A különböző platformokra való fordítás attól lesz különböző, hogy más könyvtárakban (vagy más sorrendben) fogja keresni a fordító a forrásállományokat, mindig az jut érvényre, amit hamarabb megtalál.

4.4 A platformok összehasonlítása

Hardver felépítés szempontjából a mitmót hasonlít a Berkeley mote-okra, mégis van néhány lényeges különbség. Néhány jellemző alapján összehasonlítottam ezeket a 4.2 táblázatban. Közös, hogy ATMEL ATMega128L processzort használnak, de jelentős különbség van a perifériák terén: a rádió és a rendelkezésre álló szenzorok különböznek. A Berkeley mote-okban 3 féle rádió fordul elő: az elavultnak tekinthető MICA mote-ban TR1000-es, a MICA2 mote-ban CC1000-es, a MICAz-ben pedig CC2420-as. Ezek különböző szintű kezelést igényelnek. A TR1000 a legegyszerűbb, amplitúdó-modulált adó-vevő, nincs digitális konfigurációja, a bemenete analóg, kimenete digitális, nem támogat hardveres

szinkronizációt [13]. A CC1000 már fejlettebb, saját maga is tud órajelet előállítani a bejövő adathoz igazítva és támogatja a Manchester kódolást, különálló SPI interfészen keresztül konfigurálható, de buffert nem tartalmaz, az adás és vétel bitszinten történik [14]. A CC2420 a legfejlettebb, ez már rengeteg hardveres támogatást tartalmaz, ZigBee szabvány szerinti csomagküldést és fogadást tesz lehetővé, még CRC képzést és titkosítást is végez, SPI interfészen keresztül kezelhető [15]. A mitmóthoz készült rádiós modul IA4420 típusú rádiós IC-t tartalmaz [16], melynek funkcionalitása a CC1000-es és a CC2420-as között helyezkedik el. Teljesen digitálisan kezelhető, konfigurálható, SPI interfészen keresztül; van egy kis FIFO puffere az adáshoz és vételhez, ami lehetővé teszi, hogy bájtonként lehessen az adatokat küldeni és fogadni; saját maga végzi el a küldéshez szükséges órajel előállítását (konfiguráció alapján) és automatikusan szinkronizál a bejövő adatfolyamhoz, detektálja a csomag kezdetét jelző szinkronizációs mintát is.

Platform		MICA2	MICAz	mitmót
MCU	chip	ATMega128L		
	típus	7,37 MHz, 8 bit		8 MHz, 8 bit
	programmemória	128 kB		
	SRAM	4 kB		
Szenzorkártya interfész	típus	51 érintkezős HIROSE		56 érintkezős fogalat/tűsor
	10 bites ADC	8 csatorna		4 csatorna
	Digitális I/O	8 db		16 db
	PWM	3 db		2 db
	megszakítási vonal	4 db		2 db
Rádió	chip	CC1000	CC2420	IA4420
	frekvencia	315/433/915	2400	433
	max. jelsebesség	38,4 kbaud	250 kbaud	115,2 kbaud
	interfész	SPI konfiguráció, különálló szinkron/aszinkron adatforgalom	SPI	SPI
	szolgáltatás	bitszintű küldés/fogadás	csomagszintű kezelés, titkosítás, egyéb szolgáltatások	byte szintű küldés/fogadás, beépített FIFO, automatikus szinkronizáció
Külső tár	típus	AT45DB014B FLASH		a rádiós kártyán opcionális I2C EEPROM
	méret	512 kB		
	interfész	SPI		

4.2 táblázat

A szenzorok választéka is különböző A berkeley mote-ok esetében a legelterjedtebb szenzorkártya az MST300/310 típusú [17], amely fénymérőt, hőmérőt, mikrofont, csipogót, hozzá való hangmagasság-érzékeny jeldetektort, valamint a 310-es verzió mágneses

térerősségmérőt és gyorsulásmérőt tartalmaz. A mitmóthoz kevés az igazi szenzor, illetve ami készült, az kis darabszámban lett legyártva, viszont van hozzá egy I/O kártya, amin egy digitális hőmérő mellett néhány emberi kommunikációt lehetővé tevő hasznos egység kapott helyet, konkrétan kapcsolók, nyomógombok, számkijelző és LED-ek [18].

A mote-ok programozása is más, mert a Berkeley mote-okhoz saját programozó egységeket gyártanak, ami soros, USB vagy ethernet port segítségével csatlakozik a számítógéphez, a mitmótok esetében viszont a hallgatók egy egyszerű párhuzamos portra csatlakozó programozót kapnak, valamint a laborban elérhető még néhány JTAG debugger, ami szintén használható programletöltésre. Kezdetben úgy programoztam a mitmótot, hogy a fordítás után az elkészült bináris állományt a mitmót programozására használt szoftverrel töltöttem bele a mikrokontrollerbe, azonban később megoldottam, hogy a make rendszerrel lehessen kezelni mindkét programozót.

4.5 A portolás során elvégzett feladatok

A munkám során kiderítettem, hogy milyen lépéseket kell megtenni a portoláshoz, ezeket itt összefoglalom:

- Létrehoztam a /opt/tinyos-1.x/tos/platform/mitmot könyvtárat.
    - Itt kellett elhelyezni egy „.platform” nevű konfigurációs állományt.
    - A hardware.h fájlban definiálni kellett a lábkiosztást és néhány makrót.
    - Ebben helyeztem el a platformfüggő komponensek implementációját.
- Létrehoztam a mitmot.target-et a /opt/tinyos-1.x/tools/make könyvtárban.
- Létrehoztam az stk200.extra és avrice.extra fájlokat a /opt/tinyos-1.x/tools/make/avr könyvtárban a programletöltő hardverek támogatására.
- Készítettem egy bme-hez tartozó könyvtárstruktúrát a contrib könyvtárban.
    - A hardverfüggő komponensek egy része itt helyezkedik el, méghozzá az olyanok, amelyek több, kicsit eltérő platformnál közösek lennének. A 7.3 fejezetben ismertetett marsjáró alkalmazáshoz például készül egy saját fejlesztésű processzorpanel, ami eltér a mitmóttól, de ezeken a komponenseken osztozik vele.
    - Ennek tos/lib/IA4420Radio alkönyvtárában kapott helyet a rádiós rész megvalósítása.

4.5.1 Az SPI busz

Az SPI busz a mitmót esetében több periféria kezeléséhez is kell, a gyakran alkalmazott kiegészítő kártyák esetén a számkijelző és a rádiós IC is ezen a buszon keresztül vezérelhető. Emiatt meg kellett oldani, hogy akkor se keveredjenek össze a buszon kiküldött bájtok, ha véletlenül egyszerre akarjuk kezelni a rádiót és a kijelzőt. Ehhez eddig nem volt szabványos megoldás a TinyOS-ben. A MICAz mote-ok rádióját kezelő modulban van valami nyoma annak, hogy vizsgálni kellene, hogy a busz szabad-e, de ezeknek egy része ki is van kommentezve. Egy másik dolog az SPI-vel kapcsolatban az, hogy az előbb említett rádiókezelő modul közvetlenül használja a mikrokontroller SPI-t kezelő regisztereit, de az általános interfészeket tartalmazó könyvtárban van két interfész is, ami az SPI-hez készült: a FastSPI és ByteSPI. A kettő közötti különbség az, hogy a FastSPI olyan parancsot tartalmaz, amivel el lehet küldeni egy bájtot, de csak akkor tér vissza, amikor ez végbement, a visszatérési értéke pedig az eközben beolvasott bájt lesz, míg a ByteSPI aszinkron működésű, nem várja meg a művelet befejezését, hanem egy esemény jelzi az átvitel sikerét és adja vissza a fogadott bájtot. Interfész létezik az SPI kezelésére, de egyetlen komponens sincsen az alaprendszerben, ami megvalósítaná ezeket az interfészeket. Csak néhány olyan komponens nyújt FastSPI interfészt, ami kimondottan a mote-on lévő FLASH memória kezelésére szolgál. Ahhoz tehát, hogy szépen kezelhessem az SPI buszt, meg kellett írnom a hozzá való komponenst, ez lett a HPLSpiC konfiguráció, ami tartalmazza a HPLSpiM modult. A HPLSpiC konfiguráció nyújtja a FastSPI és ByteSPI interfészeket is, valamint a BusArbitration interfészt, amivel megoldható, hogy a buszt használók elkerüljék az egymással való ütközést.

4.5.2 A busz arbitráció

Az előző részben említettem az arbitráció szükségességét, itt most a megoldást vázolom fel. Találtam néhány próbálkozást ennek megvalósítására, de egyik sem nyerte el a tetszésemet teljes mértékben. Az alapvető működéshez szükséges interfész elég egyértelmű, így máshol is megtaláltam, amire én is gondoltam: kell egy parancs a busz elkérésére (*getBus*), ami visszatérési értékével jelzi, hogy sikeres volt-e a kérés; kell egy parancs a busz felszabadítására (*releaseBus*), valamint hasznos, ha van egy esemény, ami a busz felszabadulását jelzi (busFree).

A működést úgy terveztem el, hogy a getBus hívásra a kérő megkapja a buszt, ha szabad volt, és hamis értéket kap vissza, ha foglalt volt. A használat után fel kell szabadítani,

a felszabadításról pedig mindenki értesül, az eseménykezelőben pedig már le is foglalhatja a buszt valamelyik komponens, ha akarja. Így egy tipikus buszhasználat úgy néz ki, hogy megpróbáljuk lefoglalni a buszt, de ha nem sikerül, akkor megjegyezzük, hogy mi ezt szeretnénk, és várunk a busFree eseményig. Ott sem biztos, hogy le tudjuk foglalni rögtön, ha valaki más megelőz, tehát ott is a getBus visszatérési értékét figyelve lehet, hogy várakoznunk kell tovább. Jó esetben előbb-utóbb megkapjuk a buszt, sajnos erre nincsen fix időkorlát, mert a busz lefoglalása szempontjából a monponensek közötti prioritást az fogja meghatározni, hogy milyen sorrendben futnak le az eseménykezelők. (Mivel mindegyik komponens össze van huzalozva ezzel az egy BusArbitration interfésszel, mindegyik által megvalósított eseménykezelő lefut a busFree esemény bekövetkezésekor, de ezek a kezelőfüggvények egymás után, a fordítási időben eldőlő sorrendben lesznek meghívva.) Oda kell tehát figyelnünk az éhezésre, de ha nem túlterhelt a rendszer, vannak bizonyos korlátok a buszhasználat gyakoriságára, akkor bizonyítható is lehet, hogy mindenki hozzáfér a buszhoz korlátos időn belül.

A működést a BusArbitrationC konfiguráció és a BusArbitrationM modul valósítja meg. Ezeket úgy terveztem, hogy nem csak egy, hanem több busz arbitrációját is tudják kezelni, ezért egy paraméteres BusArbitration interfészt nyújtanak, ami maximum 256 busz kezelését teszi lehetővé. A paraméteres interfész lényege, hogy egy 8 bites sorszámmal különböző interfészpéldányokra hivatkozhatunk. A sorszámot az ilyen esetekre használt „unique” makróval állíthatjuk elő. Erről itt most elég annyit tudni, hogy alkalmas arra, hogy egy közös karaktersorozat alapján a különböző előfordulási helyein 0-tól kezdve növekvően adja vissza az egész számokat, tehát minden alkalmazásakor kapunk egy egyedi azonosítót. Az SPI komponensben és mindem más arbitrációt igénylő busznál meg kell ezt egyszer hívni, hogy egy egyedi buszazonosítót kapjunk. Azért van szükség paraméteres interfészre, mert egy-egy komponensből csak egy példány létezik a rendszerben, nem lehet többet létrehozni, így több busz kezeléséhez több, különböző nevű arbitráció-kezelő komponens kellene, pedig az implementáció mindegyiknél ugyanaz.

Egy meglévő megvalósítás első ránézésre nagyon hasonló: ott szintén egy paraméteres BusArbitration interfésze van a BusArbitrationC komponensnek, viszont ott az azonosító nem a buszt azonosítja, hanem a komponens csak egyetlen busz kezelésére képes, és a buszt használóknak van azonosítója. Ez a megoldás arra jó, hogy nyilvántartja, ki kérte el a buszt és csak annak engedi felszabadítani, valamint ha ugyanaz akarja újra elkérni, aki már birtokolja, tekinthetjük sikeresnek a kérést. Én viszont nem tartottam indokoltnak, hogy ilyen nyilvántartásokat vezessek, mert gyakorlatilag nem véd semmi ellen:

- Rosszindulatúan megírt kód ellen nem lehet védekezni, mert bárhol használhatnánk közvetlenül a processzor regisztereit is, egy végtelen ciklus is elég gondot okozhat.
- Hibás kód esetén ennek ellenére is előfordulhatna, hogy úgy használjuk a buszt, hogy nem kértük el.
- Az, hogy engedi többszörösen elkérni, bajt is okozhat, ha ezt nem tartjuk nyilván, és kétszer akarjuk felszabadítani, de a két felszabadítás közben is használjuk, vagy teljesen figyelmen kívül hagyjuk, hogy már folyamatban van valami, és a két hozzáférés ütközik egymással.
- Ha egy komponens nyilván akarja tartani, akkor nyilvántarthatja, hogy sikerült-e elkérnie a buszt, és nála van-e még. Ez gyakran implicit módon jelen van egy állapotváltozóban, mert csak akkor juthatunk bizonyos állapotba, ha sikerült elkérni a buszt.

Tehát jól megírt kód esetén nem szükséges az arbitrációt végző komponensnek tudnia, kinél van a busz, hibás programnál pedig nem segít, egyedül a hiba felderítésében segíthetne, de az megoldható e nélkül is, ha valami egyszerű módon nyilvántartjuk, megkaptuk-e a buszt, és megvizsgáljuk ezt akkor, amikor a buszhoz szeretnénk hozzáférni, vagy amikor fel akarjuk szabadítani.

4.5.3 Az I2C (TWI) busz

Az I2C busszal jobb volt a helyzet, mint az SPI-vel, ehhez létezik I2C nevű interfész és volt implementációja is, csak éppen az interfészben még meg van említve, hogy „a sendStart() parancs start feltételt küld **akkor, ha szabad a busz**”, viszont az implementációban már nincs nyilvántartva, hogy szabad-e a busz vagy sem, a parancs visszatérési értéke mindig SUCCESS lesz. Tehát van javítanivaló ezen a komponensen is, de az előbbiekben leírt arbitrációs komponens megoldja ezt a problémát.

5. Egyszerű perifériák

A mitmóthoz nagy darabszámban készült a **DPY-LED-S-01b** típusú I/O kártya[18], ami főként az ember-gép kapcsolatot biztosítja: kimeneti eszközei a LED-ek, számkijelző, bevitelre pedig a gombok és kapcsolók valók. Található rajta egy digitális hőmérő is, mint igazi szenzor.

5.1 LED-ek

A LED-ek kezelése az egyik legegyszerűbb feladat volt, de ez is kérdéseket vet fel: A berkeley mote-okon 3 különböző színű LED van a processzorpanelon, míg a mintmótnál 4 egyforma (piros) a kiegészítő kártyán, valamint 1 db a processzorpanelon. A LED-ek kezeléséről a LedsC komponens gondoskodik, ami a Leds interfészt nyújtja. Ez 3 LED kezelését teszi lehetővé. Olyan megoldást kerestem, ami biztosítja a kompatibilitást a meglévő programokkal, lehetőséget ad arra is, hogy kihasználjuk a mitmót nyújtotta lehetőségeket, valamint valamennyire működik akkor is, ha nincsen kiegészítő kártya a processzormodulhoz csatlakoztatva.

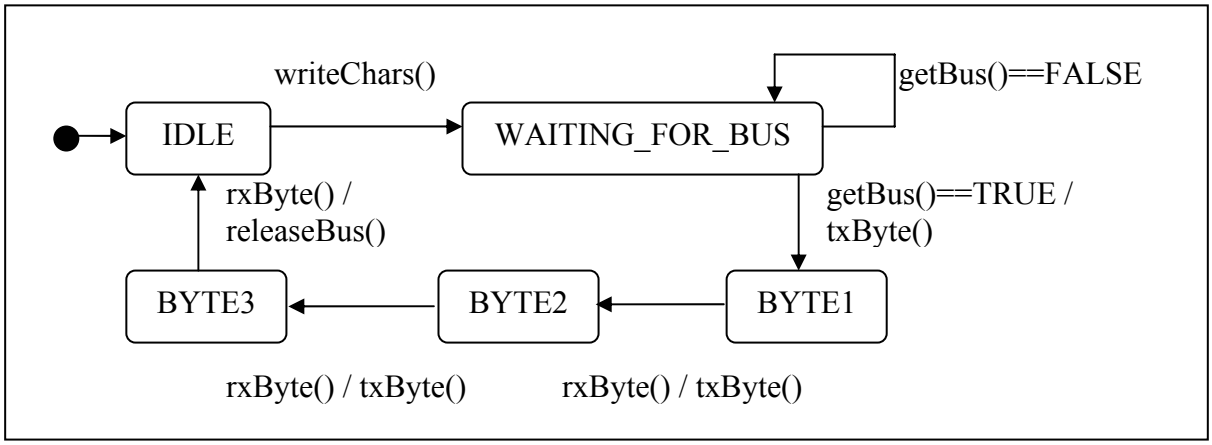
A megoldás így a következő lett:

- Az IOModule könyvtárba került egy Leds4C nevű komponens, ami az ugyanitt található Leds4 interfészt nyújtva támogat minden funkciót, amit a kártyán lévő 4 LED kezeléséhez érdemes megvalósítani: Lehetőség van a LED-ek ki- és bekapcsolására, állapotváltása egyenként külön paranccsal vagy egyetlen paraméteres paranccsal, valamint lehet az állapotot egyszerre beállítani és lekérdezni.
- Ebbe a könyvtárba került egy LedsC és LedsM komponens is, amelyek közvetítő (proxy) komponensként, a Leds interfészen keresztül biztosítják a kompatibilis hozzáférést a Leds4C komponens használatával a LED-ekhez.
- A platform/mitmot könyvtárba egy csökkentett funkcionalitású LedsC és LedsM került, ami a processzorpanelen található „SYSTEM LED”-et kezeli pirosként, a többi LED-re vonatkozó parancs pedig hatástalan. Az I/O kártya használata esetén ezeket a komponenseket az IOModule könyvtárban lévők felülbírálják.

5.2 Kijelző

Az I/O kártyán található egy 3 számjegyű 7 szegmenses kijelző, melyet 3 sorba kapcsolt shiftregiszter hajt meg. A shiftregiszter bemenete az SPI buszra van csatlakoztatva, így a kijelző tartalmát az SPI buszon 3 byte kiküldésével lehet beállítani. Természetesen eközben egy digitális kiválasztó jellel jelezni kell, hogy éppen a kijelzővel kommunikálunk, mivel az SPI busz adat- és órajel vonalán több eszköz is osztozik. A közös használat viszont egy újabb problémát vet fel szoftveres oldalról: az eseményvezérelt működés következtében nem tudhatjuk, hogy nem akarja-e több komponens is egyszerre használni az SPI buszt. Esetünkben például a rádiós modul is SPI-n keresztül érhető el. Ennek megoldására busz arbitrációt kell használni, erről bővebben a 4.5.2 fejezetben írtam.

A kijelző főként számjegyek megjelenítésére alkalmas, de szükség lehet esetleg másra is, ezért a NumDisp interfészt úgy alakítottam ki, hogy legyen lehetőség számokat is megjeleníteni, de képes legyen bármilyen szegmenskombináció beállítására is. A NumDispC modul a HPLSpiC komponens által éri el az SPI hardvert, és ez a komponens biztosítja az arbitrációt is a BusArbitration interfészén keresztül. Az SPI buszon mindig 3 bájtot kell átküldeni, ami egyben megfelel tetszőleges szegmenskombináció megjelenítésének, így a komponens működése azon alapul, hogy van egy tetszőleges 3 bájtot elküldő parancs benne. Ennek a parancsnak a végrehajtását egy egyszerű állapotgéppel végzi, ami az 5.1 ábrán látható.



5.1 ábra

Az állapotgép az IDLE állapotból indul, ahonnan a writeChars() parancs hívásának hatására kerül át a buszra várakozó állapotba. Rögtön megpróbálja megszerezni a buszt, ha nem sikerül, akkor a későbbi busFree események hatására újra próbálkozik. A lényeg az, hogy amíg ez nem sikerül, az állapota marad, viszont ha sikerül, akkor elküldi az első bájtot és átkerül a BYTE1 állapotba. A BYTEx állapot általában azt jelenti, hogy az x. bájt elküldése

van folyamatban. Ezek között akkor lépked, amikor egy bájt küldése véget ér, azaz egy rxByte esemény érkezik. SPI buszon az írás és olvasás egyszerre történik, de mivel ez egy kimeneti periféria, a beolvasott adattal nem kell foglalkoznunk. Az utolsó bájt elküldése után felszabadítja a buszt, és visszatér az alapállapotba.

A számok megjelenítése ezután már csak annyi, hogy az ezt megvalósító parancs meghatározza a szám számjegyeit, azoknak a szegmenskódját egy tömbből kiolvassa, majd ezekkel paraméterezve hívja meg a writeChars() parancsot, ami beindítja az állapotgépet.

5.3 Kapcsolók, gombok

Az I/O modulon 4 kis kapcsoló és 3 nyomógomb van, amiket főként paraméterek beállítására vagy események elindítására használhatunk. Ezek a kétállapotú beviteli eszközök a mikrokontroller egy-egy általános I/O lábára vannak rákötve, megszakítás használatának lehetősége nélkül. Így alapvetően a pillanatnyi állapotukat lehet lekérdezni, ha változást akarunk érzékelni, akkor ezt kell periodikusan megtennünk. Mivel ezek emberi beavatkozással működtethetők, a lekérdezéses megoldás is viszonylag jó megoldást ad, mert ha a feladat követelményeitől függően 10-100 ms időközönként lekérdezzük a nyomógombok állapotát, akkor a rövid lenyomások is érzékelhetők és a processzort sem terheli meg a feladat. (10 ms alatt 8 MHz órajel mellett 80 000 utasítást lehet végrehajtani, ebből a lekérdezés nagyjából 100 órajelet vesz igénybe, tehát minimális a terhelés, hacsak nem olyan alkalmazásról van szó, ahol ez is kritikusan megnövelné a fogyasztást.)
A kialakított interfész mindkét módot támogatja:

- Le lehet kérdezni a gombok illetve kapcsolók állapotát külön-külön.
- A gomboknál egyenként be lehet állítani, hogy legyen-e periodikus figyelés, és ha igen, akkor milyen időközönként. Az utóbbi esetben két esemény jelzi, ha a gombot lenyomták illetve felengedték.
- A gombok és kapcsolók funkciója egy-egy interfészen keresztül külön-külön érhető el, így ezek egymástól függetlenül hozzárendelhetők egy-egy komponenshez, aminek ilyen beviteli eszközre van szüksége. Nincs tehát a komponensben rögzítve, hogy melyik gombot használja, hanem ez a konfigurációban állítható.

5.4 Hőmérő

A panel egy LM75-ös digitális hőmérőt tartalmaz, amely I2C buszra csatlakozik. A hőmérséklet 9 biten, 0,5 ºC-os egységekben olvasható ki. A hőmérő kezelését a TempC és TempM komponensek végzik, a hőmérsékletet – mint ahogy az más TinyOS-es szenzoroknál megszokott – egy ADC típusú interfészen keresztül lehet lekérdezni.

A TempC konfiguráción keresztül lehet kezelni a hőmérőt, ez köti össze a TempM és I2CC komponenseket. A TempM modul egy állapotgépet valósít meg, állapotváltozójában nyilvántartja, hogy éppen hol tart a lekérdezés során, mert az adat kiolvasása a hőmérőből több lépésből áll:

- Meg kell szerezni az I2C busz használatának lehetőségét, várni, ha foglalt.
- Start szimbólumot küldeni, majd a hőmérő címét (írás bittel) és a hőmérsékleti regiszternek megfelelő mutatót.
- Újra kell egy start szimbólum, a cím (olvasás bittel), utána pedig 2 byte-ot olvashatunk.
- Az átvitel végén stop szimbólumot kell küldeni, majd felszabadítani a buszt.

6. Az IA4420 rádiós modul

A mitmóthoz készített **COM-R04-S01b** típusszámú ISM sávú rádió modul [19] egy Integration gyártmányú IA4420 típusú digitális FM rádiós IC-t tartalmaz [16], ennek és a rádiós kártyának az adatlapját kellett áttanulmányoznom ahhoz, hogy hozzákezdhessek a rádiós rész megvalósításához, ami a portolás legtöbb időt igénylő része volt. A 6.1 fejezetben összefoglalom az IC által támogatott funkciókat, a 6.2 részben a rádiós kommunikációt kezelő szoftver architektúráját vázolom, a 6.3 fejezetben pedig a legtöbb feladatot ellátó modul működését ismertetem.

6.1 Hardver funkciók

A rádiós IC nagyon szimpatikus, mert teljesen digitálisan, SPI buszon keresztül kezelhető. Minden SPI parancs 16 bitből áll (kivéve egyet, ahol a további órajelekre a vételi puffert lehet kiolvasni, de erre van 16 bites mód is), ez megkönnyíti a programozást.
Az IC által támogatott funkciók:

- Alapvető konfiguráció, frekvenciasáv, frekvencia, bitsebesség, vételi szűrő sávszélessége, vételi erősítés, automatikus frekvenciakövetés, adóteljesítmény, modulációs mélység, stb. beállítása
- Az IC-ben található egységek be- és kikapcsolása. Ezzel lehet az adás és a vétel között is váltani. A fogyasztást a bekapcsolt egységek határozzák meg.
- Adási és vételi puffer írása/olvasása.
- Kiegészítő funkciók beállítása: beépített időzítő, órajel-generátor, elemmerülés-jelző és kis kitöltési tényezős működés.
- Állapotregiszter olvasása
- Megszakítás kérése bizonyos események bekövetkezésekor.

A rádió nagyon sokoldalú, funkcionalitásából a következőket használom ki:

- Beállítom a kívánt átviteli paramétereket, amiket menet közben is lehet módosítani.
- Az adás és a vétel a beépített FIFO segítségével történik: Az órajelet a beállításnak megfelelően maga generálja, az elküldendő adatot egy regiszterbe kell írni bájtonként. Amikor a következő bájt fogadására készen áll, azt megszakítással jelzi. A fogadás szintén automatikus, az előhang és szinkronjel után az érkező bitekkel egy FIFO-t tölt fel, megszakítással jelzi, ha 8 bit beérkezett, ekkor a processzor ezt kiolvassa.
- A kiegészítő funkciók kezelésére is adok lehetőséget.

6.2 A kezelő szoftver felépítése

A TinyOS-ben a csomagszintű kommunikációért a GenericComm komponens a felelős. Ez egyformán tudja a rádiót és az UART-ot kezelni, mindkét kommunikációs csatornán ugyanolyan csomagok áramolhatnak. A rádió kezeléséhez a RadioCRCPacket komponenst használja, ezt kellett megvalósítani úgy, hogy az IA4420-at kezelje. A feladatot több részre bontottam, melyek különböző komponensekben kaptak helyet.

Az SPI kommunikációt a HPLSpiC komponens végzi a FastSPI és ByteSPI interfészeken keresztül. Az előbbi megszakítás nélkül dolgozik, és azonnal visszaadja a válaszul érkező bájtot, míg a ByteSPI csak elindítja a küldést, majd a 8 bit elküldése után egy eseményben értesülünk arról, hogy milyen értéket kaptunk vissza.

A közvetlen SPI kezelés fölött egy olyan réteg helyezkedik el, ami arra képes, hogy 2 bájtot küldjön el a rádiónak megbízható módon. Ennek az oka, hogy minden parancs 16 bites, mint azt korábban említettem. A megbízhatóság itt azt jelenti, hogy ha a komponens megkapta az elküldendő bájtokat, és nem volt foglalt, akkor SUCCESS értékkel tér vissza a függvény, megszerzi az SPI buszt, lehúzza a rádió kiválasztó jelét, és elvégzi mindkét bájt átvitelét. Ha az SPI busz foglalt lenne, akkor megvárja a felszabadulását, és akkor próbálkozik újra. Amikor az átvitel megtörtént, ezt egy eseménnyel jelzi, ami paraméterében tartalmazza a fogadott bájtokat is. Természetesen a kiválasztó jelet újra magasra húzza és a buszt is elengedi. Ezt a funkcionalitást az IA4420SpiC komponens valósítja meg, ami az SPI buszt a ByteSPI interfészen keresztül kezeli, így elősegíti a jó processzorkihasználtságot. Ez a réteg azért nagyon hasznos, mert így felsőbb szinten már nem kell a busz megszerzésével és a bájtok egyenkénti küldésével törődni.

Kialakítottam egy hardver absztrakciós réteget a rádiós IC számára, amit a HPLIA4420M modul képvisel, a HPLIA4420Command és az IA4420Interrupt interfészt megvalósítva. A HPLIS4420Command interfész parancsai a rádió SPI parancsainak felelnek meg, paraméterezésük illeszkedik a parancsokhoz, az IA4420SpiC komponenst használják a megfelelő parancskódok elküldésére, a parancs végrehajtását pedig a commandDone() eseménnyel jelzik, kivéve a státusz olvasást és a vételi puffer olvasását, mert ezeknél adatot is kell visszaadni, így saját visszajelző eseményük van. Az IA4420Interrupt interfész a megszakítások kultúrált kezelését teszi lehetővé, mert a rádiónak egyetlen megszakításkérő lába van, viszont sokféle oka lehet a megszakításnak, az okot a státuszregiszter kiolvasásával lehet megtudni. Ahhoz, hogy ne a magasabb szinten kelljen válogatni, hogy milyen megszakítás érkezett, a megszakítást ebben a modulban kezelem le, a kezelőrutin pedig

rögtön kiolvassa a státuszt (FastSPI interfészen keresztül éri el az SPI buszt, hogy lehetőleg ne kelljen várni és a megszakítást rögtön ki lehessen szolgálni, de szabályos buszarbitrációt végez), és különböző eseményekkel jelzi, hogy milyen megszakítás érkezett.

A legfelső réteget az IA4420RadioC alkotja, ami összeköti a komponenseket, a lényegi programkód pedig az IA4420RadioM modulban van. Ez a modul támaszkodik az előző rétegre, azaz használja az IA4420Command és IA4420Interrupt interfészeket a rádió kezelésére. Ez a legbonyolultabb komponens, de a fejlesztés során sokat javult az átláthatósága az eddig említett részek különválasztása miatt. Összetettsége miatt külön fejezetben foglalkozok vele.

6.3 Az IA4420RadioM modul megvalósítása

A GenericComm komponens a RadioCRCPacket-et használja a rádió csomagszintű kezelésére. A RadioCRCPacket konfiguráció nálam abból áll, hogy mindent átad az IA4420RadioC-nek, ami tartalmazza az IA4420RadioM-et és a működéséhez szükséges többi komponenst. Az alapvető követelmény az, hogy a BareSendMsg és a ReceiveMsg interfészt megvalósítsa, azaz tudjon üzeneteket küldeni, de közben fogadja is a beérkező üzeneteket. Tehát a rádiónak folyamatosan vételi üzemmódban kell lennie, hogy fogadhassa az érkező csomagokat, de ha küldési igény van, akkor át kell váltania adásra. Adás esetén nem árt, ha van valamilyen ütközés-elkerülés, én CSMA/CA módon oldottam ezt meg: adás előtt belehallgat a csatornába, ha nincs adás, akkor ad, ha érzékel valamilyen jelet, akkor véletlen ideig vár, és utána újra próbálkozik. A modulnak tehát le kell kezelnie azt az esetet is, ha egy csomag elküldését kezdeményezi egy másik komponens, de éppen egy beérkező csomag vétele van folyamatban. Emiatt nem célszerű egy egyszerű állapotgéppel leírni a modul működését, hanem három párhuzamosan működő állapotgéppel és néhány segédváltozóval valósítottam meg a protokoll kezelését.

Az egyik állapotgép a csomagok fogadásával foglalkozik. A rádió kezdetben inicializálatlan, de az első megszakítás egy POR (power-on reset), melynek hatására lefuttatom az alapvető inicializálást. Ezután bekapcsolja a vételi áramkört és a szinkronizációs minta felismerését, majd arra vár, hogy érkezzen valami adat. Az adatot bájtról bájtra eltárolja, majd ha a csomag befejeződött, ellenőrzi a CRC összeget, kikapcsolja a szinkronizációs minta felismerését, kiváltja a ReceiveMsg.receive() eseményt, majd kezdi újból azzal, hogy engedélyezi a mintafelismerést, ezzel lehetővé téve az újabb csomag fogadását.

A második állapotgép a csomagok küldésével foglalkozik. Alapvető állapotai:

1. Nincsen küldésre váró csomag
2. Van küldésre váró csomag
3. Véletlen várakozás folyamatban
4. Csatornába való belehallgatás
5. Vétel letiltása
6. Csomag elküldése folyamatban. Alállapotok: melyik bájtnál tart?
7. Vétel újraengedélyezése.

Az 1. állapotból a 2. állapotba a BareSendMsg.send() parancs hatására kerül, ahonnan akkor megy tovább, ha sikerült megszereznie a rádió vezérlését. Beállítható, hogy legyen-e várakozás és csatornafigyelés az adás előtt, ennek megfelelően lép tovább. A korábban leírtak alapján a 3. és 4. állapot váltogathatja egymást, ha mindkét funkció aktív és éppen foglalt a csatorna. Ha egyszer szabadnak érzékeli, akkor továbblép az 5., 6., 7. állapotokra és elvégzi a csomag elküldését, majd visszaáll az 1. állapot és jelzi, hogy a csomagküldés készen van.

A harmadik állapotgép a beállítások menetközben való módosítását végzi. Működése azon alapul, hogy néhány változóban eltárolja a kívánt beállításokat, mint például a frekvenciát, adóteljesítményt, vételi érzékenységet, és azt is nyilvántartja, hogy ezek közül mi az, ami megváltozott. Ha a módosító parancs hívásakor éppen nem lehet elvégezni a kívánt műveletet, akkor ezen változók alapján fogja végrehajtani a parancsot, amikor legközelebb alkalmas lesz.

A három állapotgép működéséhez szükséges, hogy össze legyenek hangolva egymással, mivel ugyanazt a rádiós IC-t használják, ami egyszerre egy dolgot tud végrehajtani. A modulban mindig nyilvántartom, hogy éppen melyik állapotgép az, amelyiknek joga van kezelni a rádiót, így elkerülhetőek az ütközések. Adási vagy beállítást változtató parancs bármikor kiadható, a végrehajtása viszont késleltethető, de a vétel csak akkor lehetséges, amikor éppen érkezik az információt hordozó rádióhullám, ezért a vétel prioritást élvez a másik két funkcióval szemben. A működés összehangolása emiatt úgy történik, hogy a vételt vezérlő állapotgép határozza meg, hogy mikor van alkalom a másik két gép olyan állapotváltozásaira, amely a rádióhoz való hozzáférést igényel. Ez olyankor következik be, amikor éppen mindent bekonfigurált, és vár arra, hogy csomag érkezzen. Ilyenkor a vevő kód meghív egy függvényt, ami megvizsgálja, nincs-e dolga a másik két gépnek, azaz nem várakozik-e csomag adásra, illetve nem változott-e meg valamelyik konfigurációs változó tartalma. A vétel – mint írtam – általában prioritást élvez, de ha az adásra lehetőség nyílt, akkor az adó letiltja a vevőt és elvégzi a csomag elküldését, ami

közben a vevő nem szakíthatja meg, mivel adás közben úgy sem lehetne venni, és remélhetőleg nincs is mit, ha jól működik az ütközés-elkerülés.

A modul tesztelése során nagy segítségemre volt a DiagMsg csomag, amit hibakeresésre találtak ki, és az a lényege, hogy olyan csomagokat képes rádión vagy soros porton küldeni, ami különféle típusú változók tetszőleges összeállítását tartalmazza (természetesen a méretkorlát figyelembevételével). Mivel ezt éppen a rádió tesztelésére használtam, ezért az üzeneteket nem a rádióra, hanem a soros portra irányítottam. A komponens több üzenet eltárolására is képes, arra az esetre felkészülve, amennyiben nem lenne elég gyors az üzenetek elküldése. A TinyOS komponensen kívül a MessageCenter PC-n futó javás üzenetkezelő keretprogramhoz is van egy modulja, ami képes az ilyen típusú üzeneteket emberi formátumban megjeleníteni.

A tesztelést többféle alkalmazással végeztem, először a CntToLedsAndRfm és az RfmToLeds párossal, amiket a 7.1 fejezetben is említek, majd a mitmótra kifejlesztett RfmToNumDisp és a CntToLedsAndRfm egyesítésével és kiegészítésével kapott RadioTester alkalmazással, ami erős terhelés mellett a busz arbitrációnak is jó tesztje volt, mivel egymástól függetlenül is használja a rádiót és a kijelzőt, mindkettőt SPI buszon.

7. Alkalmazások

Ebben a fejezetben néhány alkalmazást ismertetek, ami sikeresen futott mitmóton.

7.1 Tesztprogramok

A tesztprogramok arra jók, hogy megbizonyosodjunk a hardver, a fordítás és a programletöltés megfelelő működéséről, mielőtt bonyolultabb programokat tesztelnénk.

Blink:

A legegyszerűbb példaprogram, mindössze egy LED-et villogtat. Ez az első, ami a mitmóton is működött. Demonstrálja az időzítő komponens működését.

CntToLeds:

Egy számlálót tartalmaz, ami folyamatosan számol, és ennek az állása (pontosabban az utolsó 3 bit) jelenik meg a LED-eken. A LED-ek tesztelésére alkalmas, időzítőt használ, fut mitmóton is.

CntToLedsAndRfm:

Olyan, mint az előző, csak a számláló állását rádión is elküldi. Ezt használtam a rádiós modul teszteléséhez, adó oldalként.

RfmToLeds:

A rádión kapott értéket megjeleníti a LED-eken. A rádió tesztelésekor ez volt a vevő oldalon.

RadioTester:

Amikor a rádió már működött adási és vételi módban is, írtam egy olyan programot, ami a CntToRfm-hez hasonlóan periodikusan elküldi egy számláló értékét, de közben az érkező csomagok tartalmát a számkijelzőn megjeleníti. A kapcsolókkal állítható a küldés gyakorisága, aminek a mértéke a 20-100 üzenet/s tartományba esik. Ezzel a programmal lehet tesztelni azt, hogy tud-e megfelelően váltani a rádió az adási és vételi módok között, valamint sikeresen működik-e az ütközés-elkerülés. 2 mote helyett akár 3 vagy több is használható a teszteléshez. Az alapos tesztelés szükséges, mert a program fejlesztése közben volt olyan helyzet, hogy látszólag működik a rendszer, de huzamosabb idejű üzem után „lefagyott”

valamelyik mote, valamilyen kis valószínűséggel bekövetkező, nem kezelt hiba miatt. Emiatt a programot többször is átstrukturáltam, hogy átlátható és biztonságos legyen. A jelenlegi verzió alapos tesztelésen ment át, és már nem tapasztaltam megbízhatósági problémát.

TempMeter:

A hőmérő modul tesztelésére készítettem ezt a példaalkalmazást. A hőmérőt rendszeresen lekérdezi, a hőmérsékletet pedig a számkijelzőn jeleníti meg.

7.2 Távirányítású autó

Készítettem egy olyan program-együttest, mellyel egy modellautó távirányító rendszerét lehet kiváltani mitmótokkal. A modellen lévő mitmót 1 PWM és egy digitális jellel vezérli a motor sebességét és irányát, valamint egy másik PWM jellel egy szervomotort a kormányzáshoz. A távirányító oldalán 2 potenciométer szolgál a sebesség és a kormányállás vezérlésére, melyeknek az állását a mikrokonroller analóg-digitális átalakítójával határozom meg, az állás alapján pedig rádióüzenetet küldök a modellen lévő mote-nak. Az egyszerű távirányításon kívül ez a megoldás sokkal több lehetőséget rejt, mert a kommunikáció mindkét irányban lehetséges: ezt kihasználva az alkalmazásban példaként az autón lévő mote visszaküldi az általa mért hőmérsékletet, ami a távirányítón megjelenik. Természetesen más érzékelők (távolságmérő, ütközésérzékelő, iránytű, fémdetektor, …) jeleit is vissza lehetne juttatni, ezzel rögtön lehetővé téve, hogy vizuális kontaktus nélkül irányíthassuk a modellt, csak az általa szolgáltatott információk alapján. Másrészt a mote számítási kapacitása bonyolultabb irányítást, részben vagy teljesen autonóm működést is lehetővé tehet.

7.3 Marsjáró

Az I. Alkalmazott Műszaki Tudományok Versenyét Kiskunhalason rendezték meg, 2006-ban [20]. A verseny célkitűzése egy marsi analóg körülmények között helytálló marsjáró modell elkészítése volt. Csapatommal ennek megvalósításához is felhasználtuk a mitmótokat és a TinyOS-t. A roverünk egy lánctalpas jármű volt, aminek az irányítását a 7.2 fejezetbeli modellhez hasonlóan oldottam meg. A járművön lévő mote vezérelte a lánctalpakat, a talajmintavevőt és küldte vissza a bázisra az ütközésérzékelőinek az állapotát és a fémdetektor által mért értéket. (A feladat egy homok alatti vaslemez megtalálása volt egy

kövekkel tarkított pályán. A célhoz tartozott még, hogy a robot mintát vegyen a fémlemez fölötti homokrétegből.)

Ebben az alkalmazásban a bázis oldalán nem kézzel kezelhető távirányító, hanem szoftveres irányítás volt, amely egy kamera képei alapján juttatja célba a járművet. A bázison lévő mitmót a kommunikációt biztosította az irányítást végző PC és a lánctalpason lévő mitmót között, valamint még egy szervomotort kezelt, amivel egy kamerát lehetett mozgatni, panorámakép készítésének céljából. Ezt a munkát részletesen egy másik TDK dolgozatban ismertetjük a TinyOS-hez nem kapcsolódó irányítás részleteivel együtt.

7.4 Monitorozó rendszer

A tanszéken futó egyik projekt egy ökológiai monitorozó rendszer kifejlesztése, amivel egy Eger közelében lévő kísérleti erdőben kívánják leváltani a manuális mérési módszereket egy szenzorhálózattal, ami automatikusan képes elvégezni a méréseket, összegyűjteni az adatokat és GPRS segítségével az Interneten keresztül eljuttatni azokat egy szerverre, ahol azokat fel lehet dolgozni.

Éppen ilyen nagy és összetett rendszer kifejlesztésére találták ki a TinyOS-t. A nagy terület miatt multihop (több ugrásos) kommunikációt kell használni, mert nem minden mote lesz a bázisállomás olyan környezetében, hogy közvetlenül tudjanak csomagot küldeni egymás között. A másik lényeges követelmény a rendszerrel szemben, hogy energiatakarékos legyen, mert senki sem szeretne hetente elemeket cserélni az erdőbe kihelyezett mote-okban. A cél, hogy legalább 2-3 hónapig, de még jobb, ha egy éven keresztül képes a rendszer elemcsere nélkül működni. A mote-okban a fő energiafogyasztó a processzor és a rádió. Ezeknek a folyamatos működése mellett nem lehetne ilyen hosszú élettartamot elérni, ezért szükségszerű, hogy a mote-ok az idő nagy részében kikapcsolt rádióval aludjanak.

Nagyságrendi számítások alapján a processzor és a rádió összesen 40 mA áramot vesz fel működés közben, és 20 μA körül, ha alszik, így 3000 mAh kapacitású tartós elem használata esetén 1 éves élettartamra tervezve napi 11-12 percnyi működést tesz lehetővé. Ha sikerül például 1 perc alatt begyűjteni a szenzorok adatait, akkor 2 óránként végezhetünk egy-egy mérést egy éven keresztül.

Az alkalmazás úgy néz ki, hogy van egy bázisállomás, ami egy PC-vel áll összeköttetésben. Innen indul el mindig a lekérdezés, ami elárasztásos üzenetszórással terjed végig a hálózaton, és egy fa alakul ki a mote-ok között, melynek gyökere a bázis. Minden mote figyel, és várakozik a kérésre, ha pedig kapott néhányat, kiválasztja a legjobb

tulajdonsággal rendelkezőt, annak a küldőjét tekinti szülőjének a fában, majd tovább hirdeti magát, mint potenciális szülőt mások számára. Amint a kérés végigfutott a hálózaton, a fa kialakult, mindenki elvégzi a mérést, majd visszajuttatja a bázisra olyan módon, hogy a szülőjének küldi tovább a mérési eredményt tartalmazó csomagot, illetve az olyan csomagokat, amiket valamelyik gyereke rajta keresztül szeretne eljuttatni a bázisra. Mivel a rádiós kommunikáció és a mote-ok működőképessége nem teljesen megbízható a változó vételi viszonyok, ütközések és az elemek lemerülése miatt, ezért mindenki figyeli, hogy a szülőjének továbbított üzenetet a szülő tényleg továbbadja-e (ezzel a továbbküldéssel nyugtázza egyben azt, hogy megkapta a csomagot). Ha valamelyik mote azt tapasztalja, hogy hiába küldte el a szülőjének, az nem küldte tovább, akkor még néhányszor próbálkozik, de további sikertelenségek esetén kitörli a táblázatából ezt a szülőt és egy másik olyan mote-tal próbálkozik tovább, akitől valamikor kapott kérést. Ezzel a protokoll biztosítja a hibatűrőséget.

Az alacsony fogyasztást úgy oldottam meg, hogy a kéréssel együtt két időtartamot is küld a bázis: azt, hogy a kéréshez képest mennyi ideig legyenek ébren még a mote-ok, másrészt pedig hogy utána mennyi ideig aludjanak. A kérés továbbadásakor figyelembe veszik a feldolgozásokkal eltelt időt, amit levonnak az első időtartamból, így a bázistól távol eső mote-ok is ugyanakkor fognak elaludni. Az ébrenléti időt úgy kell meghatározni, hogy elég legyen az adatok begyűjtésére, az alvási időt pedig úgy, hogy a következő lekérdezéskor az órák kissé eltérő járása ellenére is mindenki biztosan ébren legyen.

7.5 Alkalmazás az oktatásban

Az elkészült rendszer már az oktatásba is bekerült, a „Szenzorhálózatok” választható tárgyból a házi feladatok TinyOS-ben elkészítendő alkalmazások voltak, a feladatok fele Berkeley mote-okon, a másik fele mitmótokon készült. Laborgyakorlatok és konzultációs alkalmak formájában segítettem a hallgatónak a rendszer megismerését. Ennek keretében megismerkedhettek a TinyOS/nesC programozás alapjaival, a rádió kezelésével, saját rádiós csomagformátum készítésével, a számítógép és a mote-ok közötti kommunikációval és a mitmót perifériáinak kezelésével. Mitmóthoz többek között készültek olyan alkalmazások, amik a számkijelző lehetőségeit bővítik ki, a gombok és kapcsolók állásában bekövetkezett változásokat regisztrálják, ami rádión lekérdezhető, valamint riasztást küldenek, ha a hőmérséklet meghalad valamilyen küszöbértéket.

8. Jövőbeli tervek

A fejlesztés nem áll meg, a további terveim között szerepel például a rádión keresztüli programfrissítés lehetőségének megteremtése. Ezt Berkeley mote-okon már megvalósították, azonban azok tartalmaznak egy-egy nagy méretű flash memóriát, amibe a programot le tudják tölteni, majd onnan átprogramozni magukat. A mitmóton nem áll rendelkezése ilyen permanens tároló, így a belső flash programmemória használaton kívüli részét kívánom a tárolásra felhasználni, így hardvermódosítás nélkül lehetővé válhat a programfrissítés. Ez a programozási mód a szenzorhálózati alkalmazások számára fontos, mert ha több tíz (vagy még több) mote-ot kell átprogramozni, az főleg a tesztelési fázisban elég sok időt venne igénybe, ha egyesével kellene a programozóra kötni. A működésre kihelyezett mote-ok esetében is lehet igény a programok frissítésére, ami sokkal kényelmesebb rádión, mintha végig kellene járni a terepet és összeszedni vagy helyben átprogramozni a mote-okat.

A TinyOS/nesC programokat szimulátorban is lehet tesztelni, nem csak a hardver-eszközbe beprogramozva, ehhez azonban meg kell írni a komponensek szimulációs verzióját. Az is a terveim között szerepel, hogy a szimulátor támogassa a mitmót platform perifériáit is.

Említettem, hogy az Eclipse nyílt fejlesztőkörnyezethez léteznek kiegészítések, amik a TinyOS alatti fejlesztést könnyítik meg a forráskódok színezésével, automatikus kiegészítéssel, könnyű fordítással, a fordítási hibák megjelenítésével. Tapasztalataim szerint az eseményvezérelt működés miatt a modulok nagy része olyan szerkezetű, hogy valamilyen állapotgépet valósít meg: vannak állapotváltozói, és az események hatására az állapotváltozók alapján kiválasztja, hogy milyen műveletet kell végrehajtani és milyen új állapotba kerül ekkor a modul. Ha a modul nagy, akkor az ilyen típusú forráskód átláthatatlanná válhat, ezt tapasztaltam a rádiós modul megírásakor. Hibát keresni nagyon nehézkes ilyen esetben. Ezeken a problémákon úgy próbálok segíteni, hogy egy grafikus programszerkesztőt készítek, amivel TinyOS komponenseket lehet szerkeszteni Eclipse környezetben. Úgy terveztem, hogy egy véges automatát vagy egy állapottérképet lehet rajzolni benne, az állapotátmenetekhez őrfeltételt és akciókat rendelni, és a forráskódot ebből az absztrakt leírásból generálja majd a plugin. A komponensek összeköttetésének létezik grafikus szemléltetése, ami segíti a megértést. Úgy gondolom, hogy a grafikus megjelenítési és szerkesztési lehetőség a program működését is átláthatóbbá tenné, nem is beszélve arról, hogy ez a reprezentáció akár a program helyességének formális vizsgálatát is lehetővé teheti.

A marsjárós alkalmazáshoz már készítettem egy .NET környezet alatti TinyOS üzenetkezelő komponenst, ami a java-s SerialForwarder programhoz csatlakozva képes

üzenetek küldésére és fogadására. Ezt is szeretném még továbbfejleszteni, hogy SerialForwarder nélkül is működjön, közvetlenül kezelve a soros portot, valamint készítenék a TinyOS-ben megtalálható Message Interface Generator (MIG) programhoz hasonló segédprogramot, amely képes a C (nesC) nyelvű üzenetstruktúrából java helyett .NET-es üzenet-osztályt generálni. Ennek a törekvésnek az az oka, hogy a .NET rendszer manapság eléggé elterjedt, a Java-hoz hasonlóan, ezért jó, ha van egy összekötő komponens a TinyOS és a .NET között. Így akár ASP.NET-es alkalmazásból, webes felületen is könnyedén kezelhetővé válhat egy szenzorhálózati alkalmazás. Készülőben van egy C++ nyelvű implementáció is erre a feladatra, ami többek között a monitorozó alkalmazás esetében teszi lehetővé, hogy a bázison csak egy kisteljesítményű ipari PC legyen, java futtatókörnyezet nélkül.

9. Irodalomjegyzék

1. ATmega128[L] adatlap:
   http://www.atmel.org/dyn/resources/prod_documents/doc2467.pdf
2. Tanszéki mitmót oldal:
   http://bri.mit.bme.hu/?l=mitmot
3. µC/OS-II honlap:
   http://www.ucos-ii.com/
4. TinyOS honlap:
   www.tinyos.net
5. David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, David Culler: ***The nesC Language: A Holistic Approach to Networked Embedded Systems***
   http://www.tinyos.net/papers/nesc.pdf
6. TinyOS Tutorial:
   http://www.tinyos.net/tinyos-1.x/doc/tutorial/
7. TinyOS Dokumentáció:
   http://www.tinyos.net/tinyos-1.x/doc/index.html
8. MICAz adatlap:
   http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf
9. MIB600 adatlap:
   http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MIB600CA_Datasheet.pdf
10. MPR-MIB users manual:
    http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf
11. TinyosIDE honlap:
    http://tinyoside.ucd.ie/
12. TinyOS Plugin for Eclipse honlap:
    http://www.dcg.ethz.ch/~rschuler/
13. RFM TR1000 adatlap:
    http://www.rfm.com/products/data/tr1000.pdf
14. Chipcon CC1000 adatlap:
    http://www.chipcon.com/files/CC1000_Data_Sheet_2_2.pdf
15. Chipcon CC2420 adatlap:
    http://www.chipcon.com/files/CC2420_Data_Sheet_1_3.pdf
16. Integration IA4420 adatlap:
    http://www.integration.com/docs/IA4420-DS.pdf
17. MTS/MDA Sensor Board User Manual:
    http://www.xbow.com/Support/Support_pdf_files/MTS-MDA_Series_Users_Manual.pdf
18. DPY-LED perifériakártya adatlap:
    http://bri.mit.bme.hu/?open=module/b/dpy_led_S_01b_adatlap_d01a.pdf
19. COM-R04 ISM sávú rádiós kártya adatlap:
    http://bri.mit.bme.hu/?open=module/1d/COM-R04-S01b_adatlap2.pdf
20. Magyar Alkalmazott Mérnöki Tudományok Versenye:
    http://www.magyarokamarson.hu