Miskolci Egyetem

Gépészmérnöki és Informatikai Kar

Elektrotechnikai- Elektronikai Intézet

Villamosmérnöki szak

Elektronikai tervezés és gyártás szakirány

UART buszrendszer és infrastruktúra tervezése

Szakdolgozat

Dankó Dávid

B35QC4

2019.

Tartalomjegyzék

1.1 Bevezetés 1

1.2 Busz rendszerek irodalomkutatása 2

1.3 EIB/KNX (European Installation Bus/KNX) 3

1.4 SPI (Serial Periferial Interface) 6

1.5 I2C (Inter Integrated Circuit) 7

1.6 1-Wire 9

1.7 Ethernet (802.3 szabványcsalád) 10

1.8 CAN (Controller Area Network) 11

1.9 UART (Universal asynchronous receiver/transmitter) 12

2 Kiválasztás és további tervezés 13

2.1 Felső réteggel szemben támasztott követelmények részletezése 14

2.2 Alsó réteggel szemben támasztott követelmények részletezése 14

2.3 Céleszközök követelményei 15

2.4 Ismeretek összegzése és a busz rendszer kiválasztása 16

2.5 Ismeretek újraértelmezése 16

3 Busz rendszer tervezése 18

3.1 Fizikai réteg (OSI-1) 18

3.1.1 Buszillesztő kerete 18

3.1.2 Első, ohmos buszillesztő 19

3.1.3 Második változat, MOSFET buszillesztő 22

3.1.4 További fejlesztési lehetőségek 23

3.2 Adatkapcsolati réteg (OSI-2) 25

3.2.1 Csomagkeretezés 25

3.2.2 Ütközéskezelés 27

3.2.3 Egyszerűsített elárasztás elleni védelem 29

3.2.4 Csomag sértetlenség vizsgálat 29

3.2.5 További fejlesztési lehetőségek 30

3.3 Hálózati réteg (OSI-3) 30

3.3.1 Hálózati üzenetszórás és csoportok (broadcast) 31

3.3.2 Hálózat címzési módszere ... 31
3.4 További rétegek (OSI-4-7) ... 34
3.4.1 Szállítási és viszony réteg ... 34
3.4.2 Megjelenítési réteg (RPC) ... 35
4 Infrastruktúra tervezése ... 38
4.1 A mikrovezérlőn megvalósított szolgáltatások ... 38
4.1.1 Felhasználói funkciók névtere ... 41
4.2 Számítógép csatolása a buszrendszerbe ... 43
4.2.1 Számítógép buszcsatoló ... 44
4.2.2 RPC szerver ... 46
4.3 CLI eszköztár ... 48
4.3.1 CLI konzol ... 49
4.3.2 CLI ping ... 50
4.3.3 CLI csomagveszteség mérés - packetloss ... 51
4.3.4 CLI ütközéses csomagveszteség mérés – collisionPacketloss ... 52
5 Összefoglalás ... 53

1.1 Bevezetés

Szakdolgozatom célja egy olyan buszrendszer és szoftver csomag elkészítése, amellyel mikrovezérlők köthetőek vezetéken össze és e buszrendszeren keresztül kezelhetőek. A buszra csatlakoztatott eszközök listája lekérhetőnek kell lennie, az egyes egységek felprogramozhatónak és képesnek kell lenniük az egymás közötti kommunikációra és így elosztott vezérlési feladatok végrehajtására. A busz rendszerbe számítógépet be lehessen csatlakoztatni, így külső parancs kiadással és a buszon lévő kommunikáció megfigyelésével magasabb szintű funkcionalitást lehessen megvalósítani.

A cél eléréséhez fontos a megfelelő buszrendszer kiválasztása (vagy kifejlesztése) illetve egy olyan hálózati réteg, amely alkalmas eszközök egymás közötti csomag átvitelre, üzenetszórásra. Továbbá távoli eljáráshívási protokoll (RPC) kiválasztása vagy kifejlesztése, amely az eszközön lévő funkciók hálózaton keresztüli meghívását teszi lehetővé. Az RPC tetejére épül az egyik legfontosabb busz képesség, a távoli felprogramozás, amely szükségtelenné teszi az eszközök közvetlen hozzáférését, így távolról, illetve az eszköz ISP kapcsainak elkötése nélkül felprogramozható.

Az ezek szerint elkészített infrastruktúra az EIB/KNX rendszerhez hasonlóan, alkalmassá válik olyan elosztott szenzor - aktuátor hálózatok kialakítására, amelyben nincs olyan központi elem, amely meghibásodásával az egész irányítási rendszer működésképtelenné válik. A funkcionalitás az eszközök egyedi felprogramozásával alakítható ki, működés közben a szenzor(ok) és aktuátor(ok) egymás közötti adatcseréjével valósul meg az irányítás.

A kutatás és tervezés során a jó kivitelezéshez szem előtt kell tartani, hogy minél kevesebb buszvezeték és mikrovezérlőn felüli külső alkatrész felhasználásával (minimális busz egység árral), minél nagyobb átviteli sebességgel és minél távolabbra lehessen kapcsolatot kialakítani.

Szoftver szempontjából fontos, hogy az egyes komponensen modulárisak, elkülöníthetőek és újrahasznosíthatóak legyenek. Mikrovezérlők esetén fontos a minimális kód méret is, így több programterület maradhat az alkalmazáskódnak.

1.2 Busz rendszerek irodalomkutatása

Ahhoz, hogy az ismert buszrendszereket össze lehessen hasonlítani és így a számunkra megfelelőt ki lehessen választani, egy közös séma mentén érdemes a busz tulajdonságait kutatni és tárgyalni. Erre megfelelő irányvonalat ad az OSI modell és annak egyes rétegein felmerülő tervezési kérdések[1].

A fizikai rétegen fontos megismernünk az átviteli közeget (pl.: vezeték, rádió, fény, hang). A csatorna használat technikáját, pl.: több vezeték, eltérő frekvenciák vagy színek használata. Az áthidalási távolságot és annak megkötéseit, pl.: áthidalási távolság függvénye a kívánt átviteli sebesség, busz vezeték kapacitása, levegő páratartalma. A zavarvédettséget, pl.: vezeték közelében lévő erős elektromágneses zaj vagy áthallás, rádió esetén a vastag épületszerkezetek és a közeli alapfrekvencián üzemelő környezetben lévő berendezések. Végül az átviteli sebesség is nagyon fontos paramétere a rendszereknek.

Mi az adatkapcsolati rétegen az átviteli közeg hozzáférésének a módja, modulációja? További tisztázandó feltételek a következők: Hogyan zajlik egy átviteli egység forgalmazása? Mi a maximális átviteli egység mérete és mekkora a keretezési többlete? Illetve tudja-e és milyen módon ellenőrzi az átviteli egységek sértetlenségét? Ha a rendszerben előfordulhatnak egyidejű adások, akkor hogyan előzi meg vagy kezeli az ütközést? Van-e mód az adások fontossági sorrend szerinti a forgalmazás során előnybe részesíteni (csomag prioritás)?

A hálózati rétegen a címzés módja: mennyi a címezhető eszközök felső határa. Támogatja-e a több eszköz vagy eszközcsoport egyidejű címzését (üzenetszórás, broadcasting)? Kezel-e több közvetetten csatlakozó hálózati réteget (útvonalválasztás, routing), illetve ha igen biztosítja-e a csomagok célba érését és az útválasztási hibák észlelését, javítását (TTL[2]).

Magasabb rétegeken (szállítás, munkamenet, megjelenítés, alkalmazás) milyen képességeket ír elő a szabvány.

[1] A rétegek és a tervezési kérdések megértésében a „Számítógép hálózatok" című Andrew S. Tanenbaum és David J. Wetherall által írt könyv segített (ISBN 978-9-635455-29-4) Ötödik magyar nyelvű kiadás

[2] Time to Live: A csomag élettartama. Egy a csomagban lévő számláló mező, ami minden útválasztásnál csökken, ha elérte a nulla értéket eldobásra kerül. A hibás útválasztás miatt fellépő csomag elárasztás elleni védelem.

1.3 EIB/KNX (European Installation Bus/KNX)

Az EIB egy olyan elosztott épületautomatizálási rendszer, amelyben a szenzorok (pl.: nyomógomb, hőmérséklet,- mozgás érzékelő) és aktuátorok (relés kapcsoló egység, dimmer, kijelző) egy közös busz hálózatra vannak kötve. Az irányítás a felprogramozott eszközök egymás közötti adatcseréjével jön létre, ebből kifolyólag a rendszernek nincs központi egysége, így egyetlen eszköz funkcionális meghibásodása nem okozza a teljes automatizálási rendszer működésképtelenségét. A hálózatba további vezérlőegységeket (pl.: logikai vezérlő összetettebb feladatok végrehajtásának vezénylésére) és rendszer eszközöket (pl.: szegmens átjáró eszköz, több hálózat összekapcsolására) lehet telepíteni.

Az EIB négy különböző adatátviteli rendszert támogat:

A „TP1" tápfeszültség vonal mellett egy további sodrott érpáron fél-duplex módban köti össze az eszközöket, differenciális jelátvitellel, 9,6 kbps sebességen CSMA/CA (Collision Avoidance) ütközésérzékelési protokollal, ami nem számít megbízható ütközés elkerülési módszernek, mivel csak az adás megkezdése előtt ellenőrzi az ütközést, így adás közbeni hibákra vagy ütközésekre nem reagál.

A „PL110" az eszközöket a tápfeszültség vonal S-FSK modulációval (szélessávú frekvenciabillentyűzéssel) 1,2 kbps köti össze. Ezáltal egy vezetékpáron az eszköz tápellátása is megoldott és a buszra is csatlakozik.

Az „RF" esetén, rádiókapcsolaton, Manchester kódolással fázisbillentyűzéssel 868,3 Mhz-en jön létre a kapcsolat.

Az „IP" esetén hagyományos ethernet kapcsolaton kerülnek az eszközök összekötésre.

Minden hálózati rétegen ugyanazt a telegram formátumot használja a rendszer.

octet 0	1	2	3	4	5	6		7	8	…	N - 1	N ≤ 22
Control Field	Source Address		Destination Address		Address Type; NPCI; length	TPCI	APCI	data/ APCI	data			FrameCheck

1. ábra KNX csomagkeret formátum[i]

Vezérlő mező (0): A csomag prioritását és az adatkeret típusát tartalmazza.

Forrás cím (1-2): csomagot feladó eszköz címe, vagy csoport azonosítója (2 byte)

Cél cím (3-4): cél eszköz címe vagy csoport azonosítója (2 byte)

Cím típusa és TTL mező (5): a címek csoport vagy eszköz azonosítót tartalmaznak, illetve hálózatok közötti útválasztás (route) esetén a TTL mező csökkentésével, ha az elérte a 0-t eldobásra kerül. Ezzel megakadályozva a hibás, hálózaton keringő csomagokat.

Adattartalom leíró (6-7): részletezi, hogy a csomagban található adat adatfolyam részlet, blokk, változó írásával vagy olvasásával kapcsolatos művelet.

Tartalom (8- N-1): hasznos adat

Adatkeret összeg (utolsó byte): ellenőrző összeg, a csomag sértetlenségének ellenőrzéséhez.

A keretformátum alapján látható hogy fix, 16 bites címzést használ, az üzenetszóró címeket leszámítva ez 61455 eszköz számára kiosztható címet jelent a hálózaton. A cím három részre bontható: [terület azonosító, 4 bit].[vonal azonosító 4 bit].[eszköz azonosító, 8 bit]

Az egyes szegmensek, azaz a vonalak a terület szegmenssel és ez a gerinchálózattal csatoló eszközökkel kerülnek összekötésre.

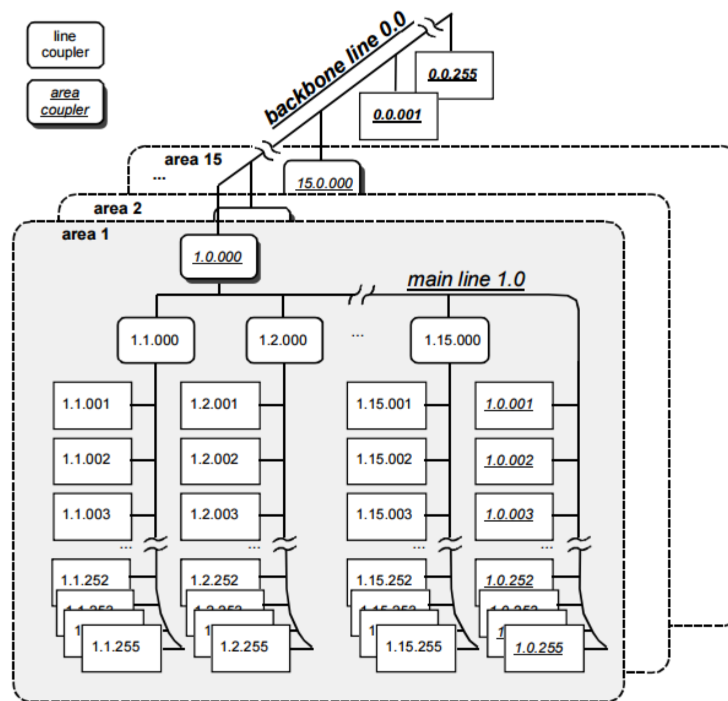
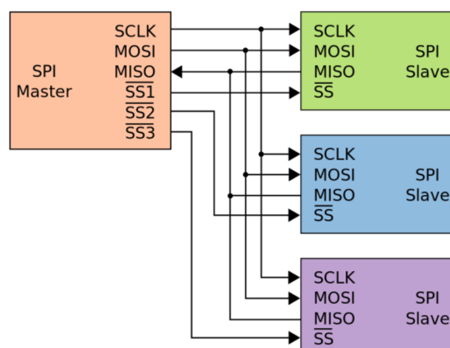


Figure 2 - The logical topology of KNX

2. ábra KNX busz szegmensei[ii]

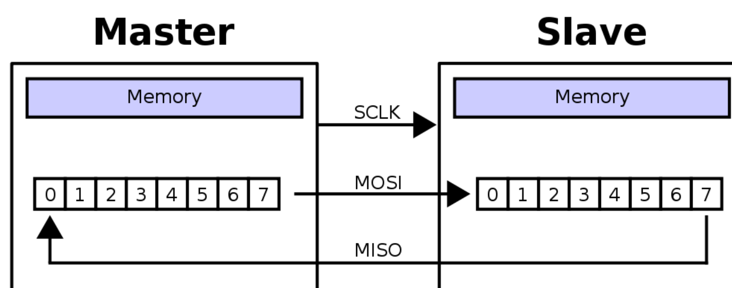
1.4 SPI (Serial Periferial Interface)

Az SPI soros periféria csatolót jelent.



3. ábra SPI egy Master és 3 Slave konfiguráció.[iii]

Beágyazott rendszerek, egyszerű perifériákkal, PCB-n belüli összekötésére tervezett, full-duplex szinkron soros, Master-slave architektúrájú egy Masteres busz rendszer. Különlegessége, hogy minimális alkatrész igénnyel (D tároló, órajel generátor) elkészíthető az adó és fogadó oldal, további bonyolultabb hardware és szoftver komponensek nélkül.



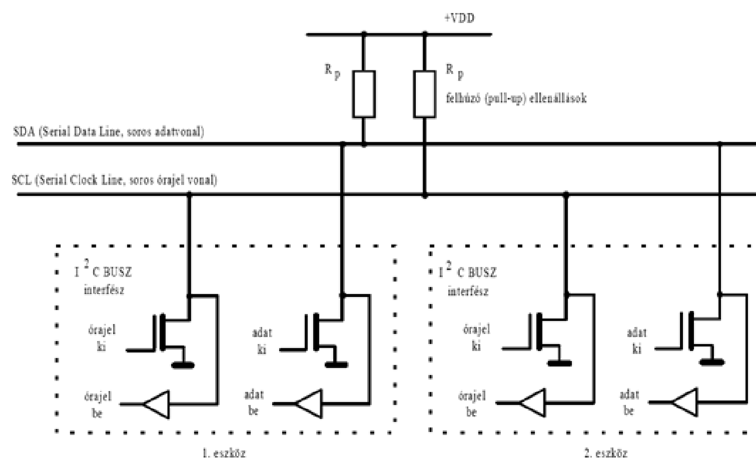
4. ábra SPI átvitel szemléltetése[iv]

A kommunikációhoz 3 vezetéket, SCLK – órajel; MISO – Master In Slave Out, Master oldali fogadó vezeték és MOSI – Master Out Slave In Master oldali adó vezeték, továbbá eszközönként egy a kiválasztáshoz CS (chip select) használ. Az áthidalási távolsága, a PCB méretein belül, legfeljebb 60 cm. Tervezésnél ügyelnek arra, hogy a távolságok és a PCB vezető sávjai olyan rövidek legyenek, hogy az ne gyűjtsön össze annyi zajt, ami megzavarja, elrontja az átvitelt, a „protokoll" nem tartalmaz csomag sértetlenség ellenőrző eljárást. Átviteli sebessége a Master órajelétől és az eszközök sebességétől függően akár több Mbites sebességgel is üzemeltethető. A Master-Slave felépítés miatt a Master eszköz adja ki a

kiválasztó jelet és az órajelet, így az átvitelek során nem fordulhat elő ütközés, de ezen felépítése miatt a Slave eszköznek nincs kontrollja az átviteli sebesség szabályozására, illetve nem kezdeményezhet tranzakciót. Ha ez utóbbira mégis szükség lenne, egy külön vezeték vonalon kell a visszajelzést megoldani, aminek a hatására a Masternek kezdeményeznie kell az átvitelt. Erre egy példa az ENC28J60 ethernet vagy az MCP2515 CAN csatoló, ami csomag érkezése esetén az „INT” megszakítást jelző lábán keresztül vissza tud jelezni a Master eszköznek. A legkisebb átviteli egysége 1 bit, a maximális átviteli egységre nincs megkötés. Ezt a buszrendszert használják a GPIO port bővítésére, SD kártyák és az előbbiekben említett külső perifériák (Ethernet, CAN) csatolásához.

1.5 I2C (Inter Integrated Circuit)

Több mikrovezérlő és egyszerű eszközök, PCB közötti, half-duplex, szinkron soros, több Master-es, csomag kapcsolt összeköttetést lehetővé tevő busz rendszer. A Philips semiconductors (Ma NXP) alkotta meg 1982-ben.



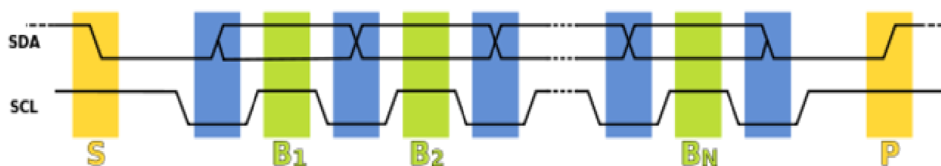
5. ábra I2C busz csatlakozás egyszerűsített kapcsolási rajza[v]

Két vezetéket vagy vezetékpárt használ. Mindkét vonal pull-up ellenállással tápfeszültségre fel van kötve, amelyet bármely eszköz le tud húzni a föld feszültség szintjére (5. ábra I2C busz csatlakozás egyszerűsített kapcsolási rajza). Az egyik vezetéken az órajel (SCL) a másikon az adat (SDA) kerül forgalmazásra. Az áthidalható távolság a busz feszültség, sebesség, illetve felhúzó ellenállástól függően 3-tól egészen 250 méterig terjedhet, de jellemző hossza 1,5 méter alatt van. A zavarvédettség rövid vezetékekkel és kisebb értékű (ohm-ban kifejezve) felhúzó ellenállások vagy sodrott érpár használatával csökkenthető. Az eredeti szabvány (1982) 100 kbit/s sebességet tett lehetővé 7 bites címzéssel. Az 1992-ben

kiadott szabvány lehetőséget adott a Fast Mode (Fm) használatára, 400 kbit/s sebességgel és lehetőséget adott a 10 bites címek használatára. A további kiadásokban nagyobb adatátviteli sebesség eléréséhez tettek kiegészítéseket.

Az átvitel szinkron, a Master az SCL vonallal szinkronizálja az átvitt adatot. Az ütközés figyelésére is ezt a vonalat használják, de ezt az adási ciklus megkezdésekor ellenőrzik. Bármelyik eszköz lehet Master a buszon, azaz bármelyik egység kezdeményezhet adást. Nincs keretezési többlet, viszont a bitek átviteléhez egy külön vezetéket (SCL) használ fel. Legkisebb átviteli egysége a byte, az átviteli egységnek nincs felső határa. Az eszközök címzése 7 bites módban 112 eszköz csatlakozását teszi lehetővé, ami 128 lehetséges címet jelent, ebből 16 cím foglalt {0000,1111}XXX. Tízbites mód esetén 896 eszköz címezhető meg, azaz 1024 lehetséges címből 128 cím foglalt {0000,1111}XXXXXX. A csomagok olvasás és írás műveleteket tartalmaznak ez a címmező utáni első bit.

A busz egyik érdekessége, hogy Master és a Slave eszköz között kézfogásos visszaigazolás lett megvalósítva, a címzett Slave (ha jelen van a buszon), a címkeret vétele után le kell húznia az SDA vonalat, ezzel jelezve a Master-nek hogy figyeli a vételt (ACK) továbbá minden egyes átvitt byte után.



6. ábra I2C egy adatátviteli ciklusának időfüggvénye[vi]

S: Tranzakció indítása a buszon, az SDA adat vonal lehúzásával.

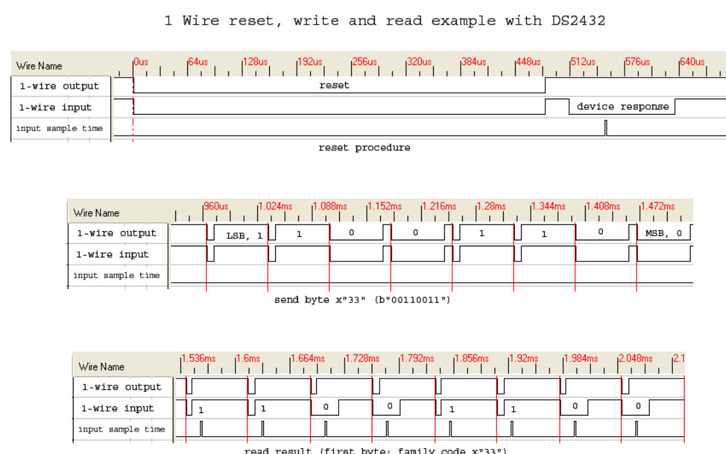
A kékkel jelölt területen a Master az SDA vezetéket a megfelelő jelszintre (átvitt adat szerint) vezérli.

B1-Bn: az SCL vonalat magas jelszintre visszaengedve jelzi a Slave-nek hogy az átvitelre szánt adat a buszvezetéken beállt a megfelelő logikai szintre és az beolvasható.

P: Tranzakció vége.

1.6 1-Wire

A Dallas Semiconductor által fejlesztett egy vezetékes egy Masteres alacsony átviteli sebességű (16,3 kbps) buszrendszer. Az Slave eszközök általában tápellátás nélküli egységek, amelyek a busz által biztosított energiából fedezik a működésükhöz szükséges energiájukat. Minden legyártott eszköz egyedi 64 bites azonosítóval rendelkezik, ezért ezek jól használhatóak azonosításra, elektronikus kulcsként.



7. ábra 1-wire adatátvitel időfüggvénye[vii]

Az időfüggvényen a „1-wire output” felirattal a Master eszköz vezérlő kimenetének jele, a „1-wire input” felirattal a buszvezeték valós jele látható, amit a Slave eszköz le tud húzni alacsony szintre. Az adatátvitelt a Master kezdeményezi. Mivel az eszközök a tápellátásukat a buszról nyerik, a sok ideig alacsonyan tartott kimenet esetén az eszköz tárolt energiája felhasználásra kerül, lemerül, ez a „Reset” a legfelső időfüggvényen. Adat küldése esetén a mester eszköz vezérli a busz vezetéket, alacsony logikai érték átvitele esetén is biztosít egy rövid időt, amíg a busz magas állapotban van (hogy az eszközök energiát nyerjenek). Vétel esetén a mester magas jelre vezérli a buszt, és a feszültség leengedésével jelzi hogy kész a következő bit fogadására. Tipikus áthidalási távolsága pár cm, de egy tanulmányban betekintés nyerhetünk nagyobb távolságú 1-Wire buszok tervezésébe. Egy másik tanulmányban több Masteres busz megvalósítását mutatják be, bár a reprodukcióhoz szükséges részleteket (forrás kód) nem biztosítják.

1.7 Ethernet (802.3 szabványcsalád)

Számítógépek összekötésére tervezett buszrendszer, mára a helyi vezetékes számítógépes hálózatok legelterjedtebb adatátviteli rendszere. Az Ethernet legelső verziója (10BASE2) vékony koaxiális kábelen (50 Ω hullámimpedanciájú RG-58) valósított meg buszforgalmat. A számítógépeket T-elosztó BNC csatlakozók segítségével lehetett összekapcsolni. A busz végeit csatlakozókkal kellett lezárni.

8. ábra Ethernet 10BASE2, T és termináló BNC csatlakozó[viii]

A busz maximális hossza 185 méter lehetett és legfeljebb 30 eszköz csatlakozhatott egymástól legalább fél méter távolságra. Ha a hálózaton valamelyik BNC csatlakozó nem zárt megfelelően, akkor az adatforgalmazás ellehetetlenült.

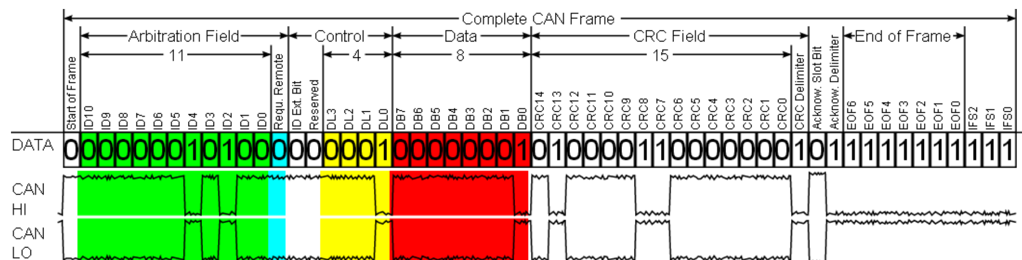
A 10BASE5 esetén vastag, 10 mm átmérőjű koaxiális (RG8) kábel került felhasználásra. Az eszközök fix telepítésű vámpírcsatlakozók segítségével csatlakoztak a buszra amelyek, a koaxiális kábel külső burkolatát felsértve a belső vezetékhez tudott csatlakozni. Ez a rendszer 10 Mbit/s sebességet garantált, maximális hossza 500 méter, legfeljebb 100 eszköz csatlakozhatott a buszhoz.

A 10BASE-T esetén (és ettől fogva a 100BASE esetén is) 8P8C (RJ45) csatlakozók segítségével kell az eszközöket egymással vagy egy hálózati eszközzel összekötni. A maximális átviteli sebessége 10 Mbit/s (a 100Base esetén 100 Mbit/s) áthidalható távolsága 100 méter. Differenciális jelátvitelt és Manchester kódolást használ az adat továbbítására. Elektronikai felhasználásra ez a változat alkalmas. Csak a drágább mikrovezérlőkbe kerül Ethernet periféria beépítésre. Ha ezzel nem rendelkezik, külső modullal lehet csatlakoztatni (pl.: ENC28J60). Annak ellenére, hogy ma már jórészt csak pont-pont kommunikációra használjuk ezt a rendszert számítógép – számítógép, illetve több eszköz esetén egy köztes eszközzel áll pont – pont kapcsolatban a gép, az eszközök még mindig támogatják a CSMA/CD ütközésérzékelési módszert. Ezáltal megoldható az, hogy egy vezetékre redukáljuk az átviteli közeget. Ütközés esetén – történjék az akár az adás közepén – a

forgalmazást az eszköz megszakítja, hogy újraadással a helyes csomagot megismételhesse. Hogy elkerüljék az újabb ütközést, az észlelés és az adás megszakítása után, de még az újraadás megkezdése előtt, véletlenszerű várakozási időt iktat be az ütközést észlelt összes fél. Mivel ez a véletlen időhossz eszközönként eltér, valamelyik eszköz megkezdi az adást a többi előtt, ezzel impliciten elnyerve a forgalmazás jogát. Tulajdonképpen a „ki húzza a rövidebbet" módszer műszaki megvalósítása, ami annak ellenére, hogy mindketten húzhatnak ugyanolyan rövidet, egyszerű és hatásos.

1.8 CAN (Controller Area Network)

A járműipar által az autókban használt buszrendszer, főleg az OBD-II elterjedése és annak iparági követelménnyé válása miatt, de használják épület és ipari automatizálási célokra is. A kommunikációt egy vezetékpáron differenciális jelátvitellel valósítja meg nagy sebességgel. Az ütközések kezelésére egy speciális technikát alkalmaz CSMA/CR (Collision Resolution). Az eddigi technikákkal ellentétben, CA (Collision Avoidance) ütközés elkerülés és CD (Collision Detection) ütközés észlelés, a CAN busz esetén, a csomag legelején egy 11 bit (kiterjesztett esetén 29) hosszúságú egyedi azonosító (Arbitration Field) található. Az adás megkezdésének pillanatában az UART-hoz hasonló módon, a start bithez igazítva minden egység megkezdi a csomag fogadását. Ha egyszerre több eszköz is elkezd adni egy időben, akkor azok az eszközök, amelyek magasabb azonosítóval rendelkeznek, a vétel folyamán olyan idő pillanatban, amikor recesszív jelet kellene adniuk, domináns jelet fogadnak, ebből azt a következtetés vonják le, hogy egy nagyobb prioritású eszköz is ad jelenleg a buszon és megszakítják saját adásukat. Ezzel nem rontják el a másik eszköz hasznos adatának átvitelét így a magasabb prioritású eszköz folytathatja tovább a forgalmazást.



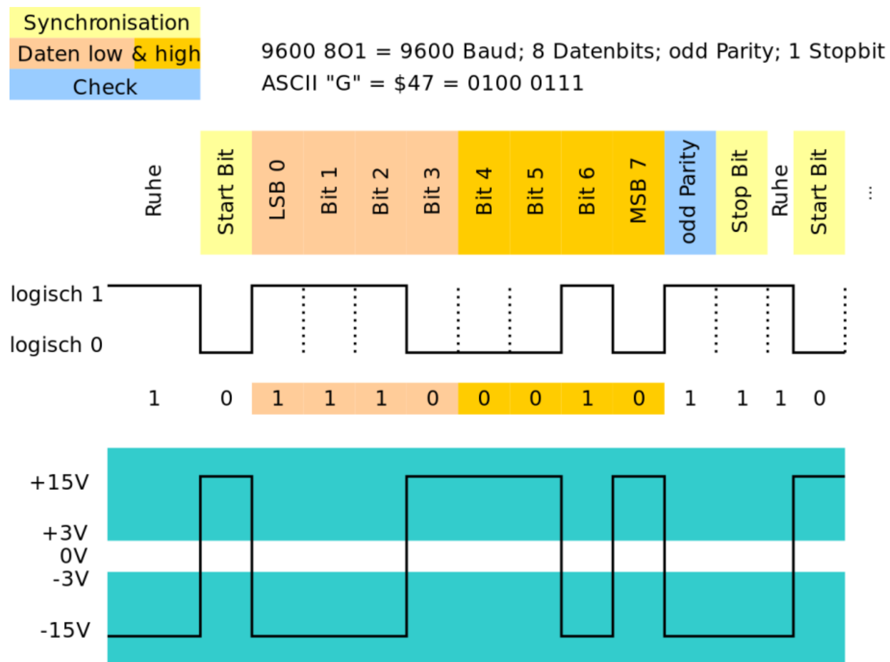
9. ábra CAN adatátvitel időfüggvénye[ix]

Az adatkeretek sértetlenségéről CRC mezőben található (Cyclic Redundancy Check) ellenőrző összeggel bizonyosodnak meg. A legmagasabb átviteli sebessége 1Mbit/s; de 512, 128, 125, 40 kbps sebességen is üzemeltethető.

Az ISO 11898-3:2006 szabvány leír egy olyan alacsony sebességű (40 és 125 kbps között üzemeltethető) hibatűrő CAN buszrendszer variációt, ami csatoló áramkörök segítségével, az egyik buszvezeték meghibásodása esetén (két vezeték közül bármelyik, de csak az egyik alacsony vagy magas logikai szintre beragad) - tartalék üzemmód gyanánt – visszavált egy vezetékes átviteli módra. A csökkentett sebesség a miatt indokolt, mivel így a tartalék átviteli módban is garantálható az adatok helyes átvitele.

1.9 UART (Universal asynchronous receiver/transmitter)

Valójában az UART (Univerzális aszinkron adó vevő) nem buszrendszer. Pont-pont összeköttetés létesítésére tervezett periféria, ami az adatokat irányonként egy vezetéken a beállított baud ráta szerinti sebességgel, továbbá két szinkronizáló bittel (Start és Stop) elküld. A vevő oldalon a start bit pontos észlelésével és a beállított baud ráta szerint a küldött biteket beolvassa, azaz annak ellenére, hogy külső szinkron vezetéket nem használ, mint pl.: az I2C, a vételt a start bittel, az adó és a vevő oldalon lévő időzítők segítségével szinkronizálják. Legleterjedtebb szabványosított formája az RS-232 ami ±15V jelszinteken forgalmaz adatot.



10. ábra UART adatátvitel szemléltetése[x]

A pont – pont összeköttetés miatt ütközés nem fordulhat elő. Így a címzés szükségtelen. A legkisebb átviteli egysége egy byte erre még 3 további bit adódik, mint keretezési többlet (overhead).

Szinte minden egyes mikrovezérlőben, számítógépben megtalálható ez a periféria.

Tárgyalására – mint ahogy az a szakdolgozat címéből sejthető - csak azért került sor, mert később ennek a perifériának az ismerete kulcsfontosságú lesz a szakdolgozat kidolgozásánál.

2 Kiválasztás és további tervezés

Az elterjedt, forgalomban lévő buszrendszerek ismertetése után ki kell választanunk azt, amelyet a rendszer legalsó rétegének válasszjuk és továbbiakban ennek figyelembe vételével tervezzük meg az erre épülő további rétegeket. Ehhez azonban jobban meg kell fogalmaznunk, hogy pontosan milyen funkcionalitást várunk el a kész rendszertől és a kiválasztást ennek tudatában kell megtenni.

2.1 Felső réteggel szemben támasztott követelmények részletezése

A mikrovezérlőkbe szánt szoftvernél fontos, hogy a buszrendszerre különböző gyártó különböző modellű eszközei egységes módon csatlakoztatható legyen és a később kidolgozott és részletezett alapfunkciókat minden eszköz meg tudja valósítani.

Az alap funkciók a mikrovezérlő szoftverében, egy - egy meghívható funkcióként kerülnek megvalósításra.

Ezeket a funkciókat egy kis kódlábnyomú RPC (Remote Procedure Call) kiszolgálóval lehet a legegyszerűbben elérhetővé tenni. A technika lényege, hogy a hálózatról érkező csomagot diszpécseléssel a mikrovezérlőben lévő olyan funkcióhoz továbbítjuk, amely a beérkezett csomagból a célfunkció paramétereit ki tudja csomagolni, azt meghívni és a választ becsomagolva vissza tudja küldeni a hívást indító félnek. Ennek az alrendszernek a kiválasztása (vagy megvalósítása) is egy külön feladat.

Ezeket a funkciókat, a kezelhetőség és az intuitív kezelés érdekében, érdemes a modern programozási nyelvek mintájára névterekbe szervezni. Ezáltal elkerülhetőek a globális névtérben ömlesztett funkciók karbantartásának kellemetlen feladata, illetve ez a minta bővíthetőséget biztosít, amely lehetőséget az alkalmazás fejlesztőjének is biztosíthatunk. Az így elérhető funkciókat a mikrovezérlők egymás között is meg kell tudni hívni.

Ha az üzenetszórási mechanizmushoz a kiválasztott buszrendszeren jobban hozzáférünk, kialakíthatunk az eszközcsoportok kezelésére alkalmas keretet is. Így egy lekérő vagy irányító paranccsal több, azonos feladatot ellátó egységről is lekérhető információ, vezérelhető.

2.2 Alsó réteggel szemben támasztott követelmények részletezése

Ahhoz hogy a felső rétegben ismertetett követelményekhez megfelelő alapot biztosítsunk, olyan busz rendszerre van szükség, ami multi Master felépítésű, hiszen csak ez alkalmas arra (Master-Slave-vel való összevetésben), hogy bármely két tetszőleges eszköz közvetlenül csomagot küldjön egymásnak.

Ahhoz hogy az eszközök könnyedén felfedezhetőek legyenek, a busznak üzenetszórási módszert is támogatnia kell. Ez a funkció hasznos az eszközcsoportok kialakításához is, de az előző pont miatt feltétlenül szükséges.

Az egységek teljes ára úgy minimalizálható, ha minél kevesebb és minél egyszerűbb külső alkatrészt használunk fel. Illetve, hogy csökkentsük a mikrovezérlő terheltségét, a közvetlen *bit-banging*[3] technikákat érdemes mellőzni. Mindkét szempont azt sejteti, hogy érdemes olyan perifériát előnybe részesíteni, ami a legtöbb mikrovezérlőben megtalálható.

A fizikai busz szempontjából fontos hogy, minél kevesebb vezetéket használjon, mivel a több vezeték növeli a busz vonalának kialakítási költségét, illetve többletfeladatot jelenthet a több vezetéken való hibakeresés és elhárítás.

Végül, de nem utolsó sorban, legyen a busz képes minél nagyobb átviteli sebességre. Azon felül, hogy így alkalmunk adódik nagyobb adatterhű feladatok elvégzésére (több parancs csomag, illetve ezek mellett adat csomagok forgalmazására is), a nagyobb átviteli sebesség rövidebb csomag átviteli időt is jelent, azaz közvetetten rövidebb a buszeszköz válaszideje.

Céleszközök követelményei

A célkitűzésben az eszköz gyártójára, illetve eszközcsaládjára nem lett utalás téve. Ez szándékos, az általunk támasztott követelmény szerint a kész rendszerbe több különböző eszközcsalád gyártmányát is be kell tudni építeni. Az eszközök közötti különbség legfeljebb a részletekben szabad felfedezni.

Például: a kódfeltöltés folyamata minden eszköz esetén megegyezik, viszont a feltöltendő forráskód lefordításához le kell kérnünk az eszköz családját és modelljét. Ez alapján a hozzá megfelelő fordítóval az alkalmazás forráskódját le tudjuk fordítani, az így létrejött bináris állományt az egységes kódfeltöltési folyamattal fel tudjuk a céleszközre küldeni.

Ezek alapján a céleszközök kiválasztásánál fontos, hogy az eszköz képes legyen a saját kódterületén lévő adatokat módosítani, azért hogy a programkódot üzem közben, a buszról kapott adatokkal ki lehessen cserélni.

[3] Célhardver helyett szoftveres úton megvalósított adatátvitel. Általában a GPIO portokkal és a pontos időzítést és jelváltozást érzékeléséhez aktív várakozást használva. Amely az átvitel idejére a CPU teljes lefoglaltságát eredményezi.

Alapfeltétel hogy az eszköz rendelkezzen a busz által használt perifériával, vagy egy csatlakoztató perifériát lehessen hozzá kapcsolni. Ez utóbbi legtöbb esetben SPI-on keresztül valósul meg.

2.3 Ismeretek összegzése és a busz rendszer kiválasztása

Az előző részben ismertetett pontokon végighaladva és kiértékelve az találjuk, hogy a követelményeink ellentmondóak, praktikusan teljesen nem megvalósíthatóak.

Az UART periféria szinte minden mikrovezérlőben megtalálható, de valójában nem buszrendszer. Pont-pont összeköttetést valósít meg, így közvetlenül nem használható fel. A multi Master szükségessége miatt az SPI, és a 1-Wire nem használható. Így csak az I2C, CAN, ETHERNET, valamint egy kutatásban említett multi Masteres 1-Wire, amelynek a megvalósítása (program kódja) nem áll rendelkezésemre.

Ezeket összevetve, a multi Master 1-Wire-nek nem áll rendelkezésre a forrása és a bit-banging technika alkalmazása miatt elvetésre kerül, mivel a maximális átviteli sebessége 16,6 kbaud/s és emellett az adás és vétel ciklus alatt teljesen lefoglalja a mikroprocesszor CPU idejét. Bár ez utóbbi enyhíthető lenne megszakításos üzemmel és időszakos mintavételezéssel. Az előbbi 3 buszrendszerből a CAN és az Ethernet nem kifejezetten elterjedt a mikrovezérlők perifériái között, ezért kisebb egységek esetén külső csatolókat kell alkalmazni (pl.: Ethernet esetén ENC28J60, CAN esetén MCP2515) viszont csak egy vezetékpárt használnak differenciális üzemben az átvitelre. Az I2C viszont szinte minden mikrovezérlőben megtalálható, ellenben 2 pár buszvezetéket használ a kommunikációra.

A dilemma az, hogy vagy beépített perifériát használva plusz vezetéket használunk a kommunikációra vagy közel megkétszerezzük az egy vezérlőre jutó költséget a külső perifériák szükségessége miatt. Bármelyiket is választjuk, a kitűzött célok közül nem tudunk mindent teljesíteni.

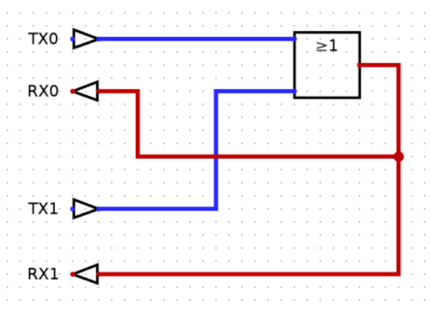
2.4 Ismeretek újraértelmezése

A dilemma eldöntése helyett tovább kutattam. Milyen megoldások lehetségesek az elterjedt mikrovezérlők busz kommunikációjára? Esetleg van-e olyan mikrovezérlő a piacon amibe CAN vagy Ethernet van integrálva és a költsége nem sokkal nagyobb, az eddig ismertetetteknél.

Azonban ekkor a kutatás új irányt vett: Új eszközök keresése helyett végignéztem, hogy az eddig általalam ismert mikrovezérlőben milyen perifériák vannak. Az SPI az I2C és az UART szinte minden nem minimalizált felszereltségű (pl.: ATtiny és PIC12F sorozatú) vezérlőben megtalálható.

Az UART felhasználása került újra szem elé. Ez egy pont-pont kommunikációra tervezet periféria, de ha jobban szemügyre vesszük, valójában „csak” egy szinkron adó-vevő páros, amihez nincs hardveres címzési módszer vagy bármilyen protokoll. A eszközökben egy byte átviteli egységre van kialakítva adó és vevő buffer. Átviteli sebességük konfigurálható és általában megszakítási vonallal is el vannak látva.

Egy további egyszerű tapasztalat az, ha a vezérlő adó (TX) és vevő lábát (RX) összekötjük, akkor a kiküldött adat a vevő bufferben megjelenik. Ha két vezérlő TX lábát „vagy” (fordított logika esetén „és”) kapun keresztül összekötjük és ennek kimenetét visszavezetjük a vevő lábakra (RX0 és RX1) az (11. ábra UART közösített átvitel tesztje) látható módon akkor az egyik eszköz által elküldött adat megjelenik a küldő eszköz és a másik eszköz vevő bufferben is.



11. ábra UART közösített átvitel tesztje

Azaz ilyen módszerrel az UART használható több eszköz közötti adatátvitelre.

Ennek ismeretében kialakítható egy olyan buszrendszer, ami az UART perifériát használja, ami több szempontból is bizakodásra ad okot: az UART a legtöbb eszközben megtalálható és a legtöbb eszköz esetén vezérlőkhöz mérten nagyobb átviteli sebességre képes (akár 1 Mbps sebességet is képesek elérni). A beépítettsége miatt, mivel a vezérlők belső buszrendszerére vannak az UART vezérlőregiszterei kötve, az átvitt adatot nem kell még egy külső perifériáról beolvasni (mint pl. külső CAN vagy Ethernet csatoló esetén) azaz a buszról érkező adatok a RAM-ba gyorsan átmozgathatóak.

Ez egy nagyszerű lehetőség, aminek a választása azt a következményt vonja maga után, hogy egy alapvetően adatfolyam átvitelre használt perifériából kell csomag alapú átviteli

rendszert készíteni. Ám ennek kezelnie kell az irodalom kutatásban, a buszrendszerek által ismertetett képességeket, mint például a címzés, az ütközés észlelés, adatkeret sértetlenség ellenőrzés.

Az előbbiekben ismertetett dilemma fényében ez a lehetőség túlságosan vonzó ahhoz, hogy veszni hagyjuk. Így a döntés egy új UART alapú buszrendszer kidolgozása lett.

3 Busz rendszer tervezése

Ahhoz hogy egy új buszrendszert megvalósítsunk, az irodalomkutatásnál említett, az egyes OSI rétegek tervezési kérdéseire kell kielégítő választ adni, ezek kerülnek most kifejtésre.

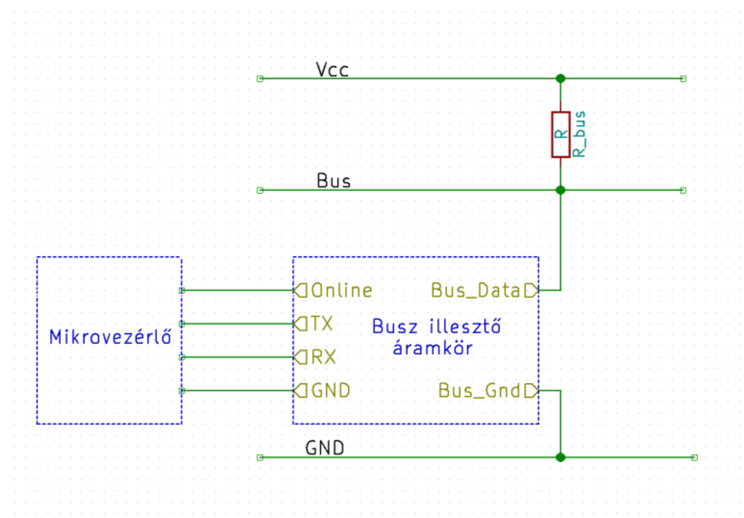
3.1 Fizikai réteg (OSI-1)

Milyen közegen szeretnénk az eszközöket összekötni? Hogyan jön létre a csatlakozás a közeghez?

A cél az, hogy egy vezetékpár használatával az összes busz eszköz össze lehessen kötni, mint a CAN vagy az Ethernet esetén. Az előző pontban ismertetett „vagy" kapuval összekötött eszközöket úgy le kell egyszerűsíteni, hogy ne kelljen minden egyes eszközhöz egy közös vevő és minden eszközhöz külön adó vezetéket kötni. A cél olyan busz illesztőt létrehozni, ami a kiküldött adatot egy vezetékpáron továbbítja minden eszköznek és mi magunk is megkapjuk a küldött csomagot.

3.1.1 Buszillesztő kerete

Itt segít az I2C és a 1-Wire busz ismerete, amelyek egy vezetékpárt használnak (GND és adat), ahol az adat vezeték egy felhúzó ellenállás segítségével, tétlen állapotban aktív feszültségszinten van. Ezt a sémát felhasználva az egy vezetékpár használata kivitelezhető.

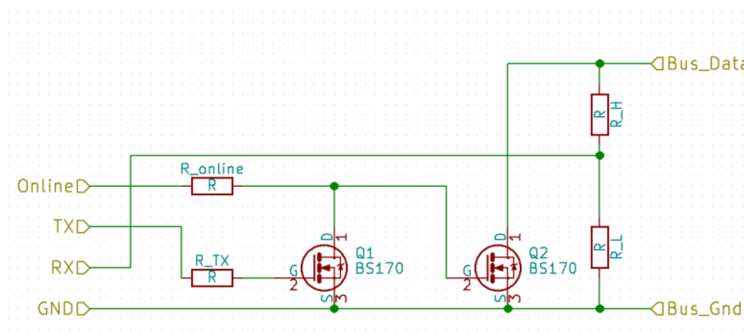


12. ábra Mikrovezérlő csatlakoztatás a buszra

Az ábrán egyedül az „Online” láb szorul némi magyarázatra. A vezérlő áramkör csak akkor kerül aktív állapotba, ha ezen a lábon logikai magas szint jelenik meg. Erre azért van szükség, mivel ha az UART port nem kerül aktiválásra, vagy az eszköz épp külső felprogramozás alatt van, és emiatt a kimenetei Resetelve vannak, ezáltal az UART kimenet helytelen állapotba kerül, így az megbéníthatja a busz kommunikációt. Illetve további haszna van, ha a buszra való felcsatlakozást vissza tudjuk tartani.

3.1.2 Első, ohmos buszillesztő

Az első buszillesztő áramkör feszültség osztóval csatlakozott a buszra



13. ábra Ohmos buszillesztő

A kapcsolás tulajdonképpen egy szintillesztő, ami lehúzó áramkörrel van felszerelve. Általában az UART tétlen állapotban magas logikai szintre van húzva, ezért kell egy invertáló tag az illesztőbe, ami a „Q1” N-MOSFET-tel lett megvalósítva.

A buszrendszer fejlesztésének korai fázisában ezt az illesztőt használtam a minimálisan működő verzió elkészítéséig, ekkor még csak 19 kbps sebességen fejlesztettem.

Ennek az illesztőnek a szemmel látható hátránya, hogy tétlen állapotban hozzájárul a busz fogyasztásához mivel R_L és R_H ellenállásokon keresztül (R_{bus} busz vonal felhúzó ellenálláson keresztül) áram folyik a busz adat és föld vezetéke között.

Méretezése: R_{TX} és R_{online} ellenállásokat 1kΩ-nak választottam, ez többszörösen elég ahhoz, hogy a mikrovezérlő kimenetét gyors kapcsolás esetén se terhelje túl, illetve elég kis értékű ahhoz a gate kapacitást (adatlap szerint 60pF) gyorsan feltöltse. Egy bit ideje a maximális, 1Mbps sebességen (1)

$$t_{bit} = \frac{8bit}{8*1Mbit} = 1\ \mu s \quad (1)$$

TX láb maximális áram terhelése (2)

$$I_{\max} = \frac{5V}{1\ k\Omega} = 5mA \quad (2)$$

A TX láb legtöbb esetben legalább 25 mA-re vannak méretezve[4].

MOSFET bekapcsolási ideje 1 kΩ ellenállás esetén első közelítésben (3)

$$t_{gate} = 1k\Omega * 60\text{pF} = 60\ \text{ns} \quad (3)$$

Végül R_L és R_H arányát úgy kell megválasztani, hogy a busz tétlen állapotban, a feszültség osztón keresztül se haladja meg az RX bemenet tolerancia feszültséget $U_{RX_{max}}$-ot.

$$U_{bus} * \frac{R_L}{R_L+R_H} < U_{RX_{max}} \quad (4)$$

Így az egy eszközre jutó fogyasztás tételen állapotban:

$$P_{idle} = \frac{{U_{bus}}^2}{R_L+R_H+R_{bus}} \quad (5)$$

[4] Adatlapok „Absolute Maximum Ratings” alapján a legkisebb:
ATmega168 („DC current per I/O pin”: 30 mA),
ATmega328 („DC current per I/O pin” 40mA),
STM32F103C8T6 („Output current sunk by any I/O and control pin”: 25 mA)
PIC18F2550 („Maximum output current sunk by any I/O pin”: 25 mA)

A busz teljes fogyasztása n eszközzel:

$$P_{idle}\ n\ = \frac{{U_{bus}}^2}{\frac{R_L+R_H}{n} + R_{bus}} \qquad (6)$$

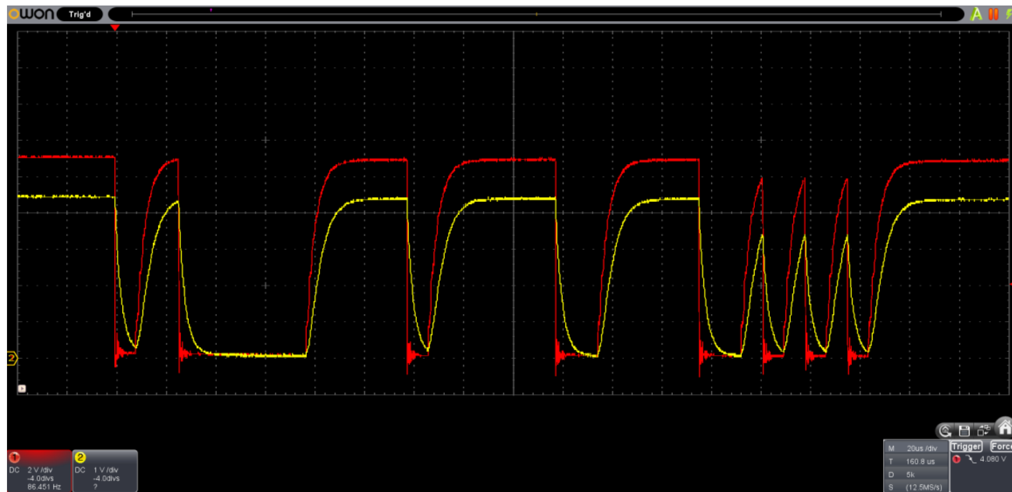
Végül „n” darabszámú eszköz felcsatlakoztatásának a feszültség csökkentő hatása az RX lábakon:

$$U_{RX_{max}}\ n\ = U_{bus} * \frac{R_L}{R_L+R_H} * \frac{\frac{R_L+R_H}{n}}{R_{bus}+\frac{R_L+R_H}{n}} \qquad (7)$$

A kezdeti fejlesztésnél R_{bus} értékét 500Ω, R_L értékét 100 KΩ, R_H értékét 147 kΩ értékre választottam illetve 12V U_{bus} feszültségre illesztettem.

Ezekkel az értékekkel végigszámolva 230 busz eszköz esetén $U_{RX_{max}}$ feszültsége még mindig 3,3V felett van (az UART portok általában 3,3V-ra vannak méretezve, de 5V toleránsak), illetve 0,1 W az egész busz fogyasztása. Ami ilyen eszközpark mellett nem számottevő.

Ez R_L és R_H ellenállások nagy értékűre való megválasztásával enyhítető, ám ekkor egy másik problémába ütközünk: a mikrovezérlők RX lábának kapacitása csökkenti a jelváltozás sebességét, így a jelet olyannyira eltorzítja, hogy az nagyobb csomag veszteséghez vezethet vagy a vétel teljes ellehetetlenüléséhez.



14. ábra Ohmos buszillesztő időfüggvénye 115 kbps sebességen

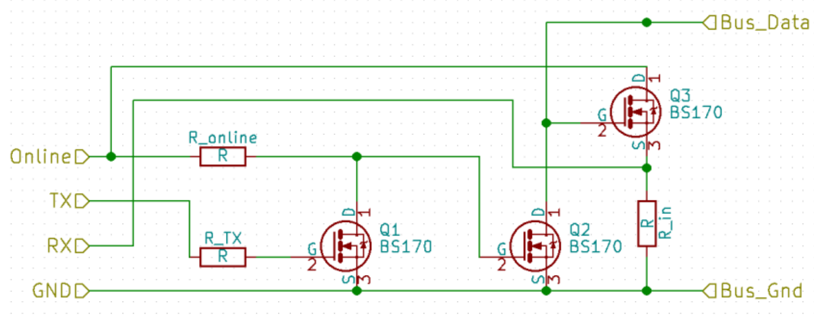
A 14. ábra egy csomag adatforgalmát mutatja be 115 kbps sebességen. Az egyes, piros színnel rajzolt csatorna a buszvezeték, amely tétlen állapotban 500 Ω ellenálláson keresztül 12V feszültségre van felhúzva. A kettes, sárgával rajzolt csatornán a mikrovezérlő

RX bemenetén mért jelalak látható. Jól látható hogy több egymás utáni 1-0-1-0 váltás esetén a jel nem tud az eredeti feszültségig visszatérni (nem tud az RX láb kapacitása a nagy ellenállásokon keresztül feltöltődni)

Ez a hatás a csatoló ellenállás növelésével (R_L és R_H), más nagyobb RX bemeneti kapacitással rendelkező eszköz esetén, illetve nagyobb átviteli sebesség esetén még számottevőbb.

3.1.3 Második változat, MOSFET buszillesztő

Az adatátviteli sebesség növelésével (az első fejlesztési ciklusban említett 19 kbps sebességről 115 kbps-re váltva) megnövekedett a hibás csomagok száma. A 11. ábrán a jelenség oka jól megfigyelhető. Világossá vált, hogy nagyobb sebességhez valamilyen vevő oldali illesztésre van szükség. Felmerült, hogy olyan komparátor felhasználásával kerüljön az RX a buszvezetékhez illesztve, ami tolerálja a buszfeszültséget és a referencia lábán lévő feszültséggel hangolható a vétel minősége. Ám közben az adatlapokat újra áttekintve azt a felfedezést tettem, hogy az általam használt BS170 U_{GS} feszültsége $\pm 20V$ toleráns. Mivel 12V-os buszfeszültséget használok, ez azt jelenti, hogy közvetlen illeszthető a buszvezetékre.



15. ábra N-MOSFET buszillesztő

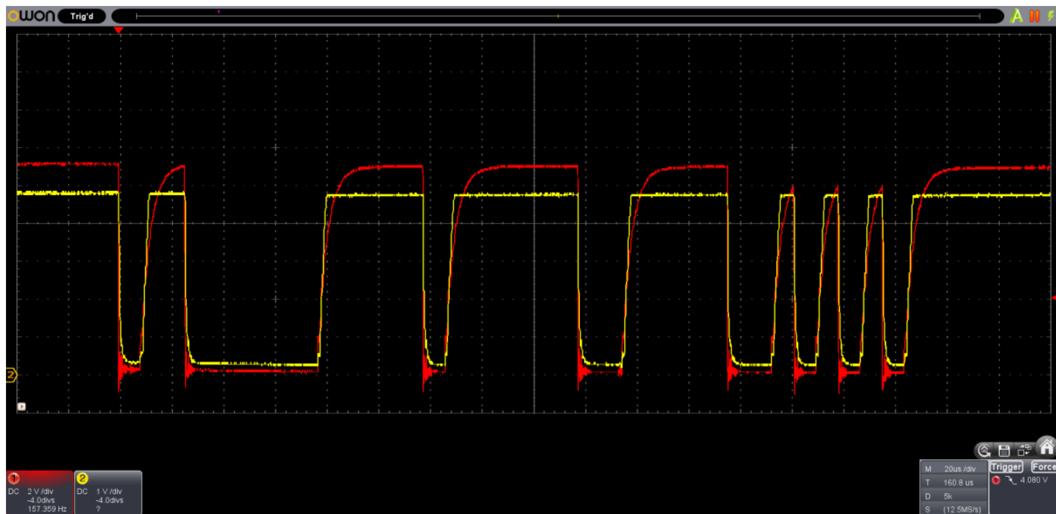
Ez kiküszöböli az ohmos csatolónak a fogyasztással kapcsolatos problémáját, mivel így csak az áttöltés ideje alatt történik a buszeszköz részéről a busz irányába teljesítmény felvétel (nyilván a logika alacsony jelű periódusú alatt, a FET az R_{bus}-on keresztül az U_{bus} feszültségét lehúzva ekkor is fogyaszt).

Viszont erősítőként is működik, ezáltal a busz eredeti felfutási sebességénél élesebben fut fel az RX lábra kötött feszültség. Az R_{in} ellenállásnak az RX lábon lévő

feszültséget kell lehúzni, azaz a láb kapacitását kisütni, amikor a buszon kis alacsony logikai szint van jelen és a FET bezár.

Ennek méretezése az Ohmos illesztőnél tárgyalt módszer szerint történt, 1kΩ.

Magas logikai szint esetén a Q3 FET kinyit és az „Online” kapcson lévő feszültséget rákapcsolja az RX lábra.

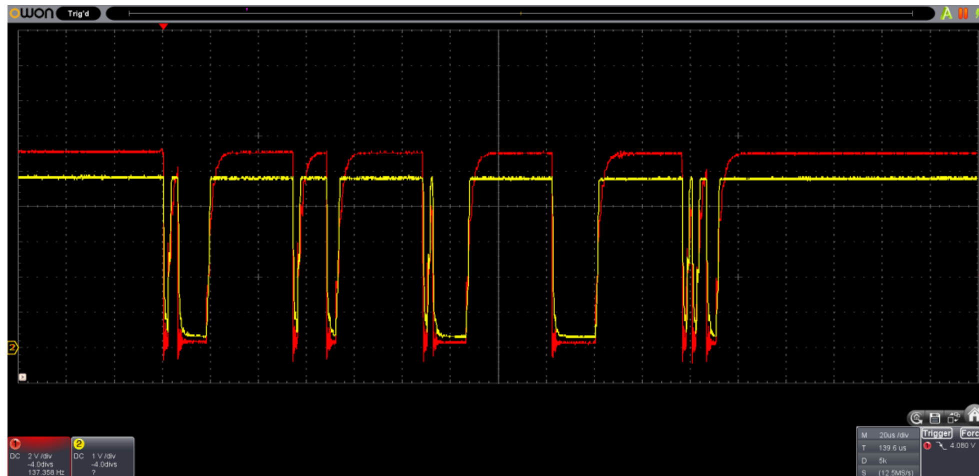


16. ábra MOSFET buszillesztő időfüggvénye 115 kbps sebességen

Végeredményben ez sokkal biztosabb jelalakot biztosít és ez által kisebb csomagvesztési arányt, egy egyszerű és már használt alkatrész újbóli felhasználásával.

3.1.4 További fejlesztési lehetőségek

A fejlesztést jelenleg 115 kbps buszrendszeren végzem. A bemutatott mérések 120 méter UTP sodrott érpáron történtek. Nagyobb sebességek eléréséhez valószínűleg új, más típusú buszillesztők fejlesztésére lehet szükség, pl.: differenciális jelátvitelre, mint a CAN esetén.



17. ábra Busz feszültségének időfüggvénye MOSFET illesztővel 500 kbps sebességen

Továbbá elvben átültethető a buszrendszer olyan közegre ahol az eszközök által kibocsátott jelek „vagy” kapcsolatát képezhetjük. Vezeték esetén a „A” eszköz vagy „B” eszköz húzza le a buszfeszültséget jelentette ezt az összefüggést. Ezek alapján LED - fotocella vagy rádió adó-vevő párok felhasználásával is kialakítható, bár ezek nem olyan megbízhatóak, mint a vezeték jelentette megoldás.

A MOSFET csatoló esetén felmerült az a hibalehetőség, hogy ha a buszvezeték valamilyen okból ideiglenesen 20V feszültség fölé szökik viszonylag hosszabb időtartamra (a BS170, 50 µs időtartamig elviseli a ± 40V feszültséget is) akkor a gate - source átmenet átütésével ez a buszfeszültség megjelenhet az RX lábon, ami valószínűleg a mikrovezérlő tönkretételével járna. A megoldást egy biztosíték és egy szupresszor áramkör beiktatása jelentené, amely túlfeszültség esetén a buszvezeték és a föld rövidre zárásával kioldaná a biztosítékot, ezzel megóvva a buszeszközt. A biztosíték helyére visszaálló polimeres biztosítékot lehetne használni, ami a hiba megszűnése után egy bizonyos idővel újra vezetési állapotba került. A szupresszor egy, a buszfeszültség fölé méretezett Zener diódával vagy kifejezetten erre a célra tervezett szupresszor diódával lehetne legegyszerűbben megoldani, ám a Zener diódával parazitikus kapacitása lassíthatják a busz jelváltozási sebességét. A meghibásodással szembeni ellenállása további kutatás, méretezés és kísérlet tárgyát képzi.

A differenciális jelátvitelre való tervezésnek egy oldalága a hibatűrő átvitelre való tervezés a CAN busznál említett speciális meghajtó áramkör felhasználásának segítségével, amely a differenciális adást az egyik buszvezeték meghibásodása esetén a jelenlegi átviteli

módszerhez hasonló módba valósítja meg a vezetékek vezérlésének adaptív megváltoztatásával.

3.2 Adatkapcsolati réteg (OSI-2)

Hogyan küldjünk és fogadjunk csomagot a hálózaton? Hogyan észleljük és kezeljük az egyidejű adások ütközését? Hogy akadályozzuk meg a csatorna elárasztását? Honnan tudjuk, hogy a fogadott csomag hibátlan?

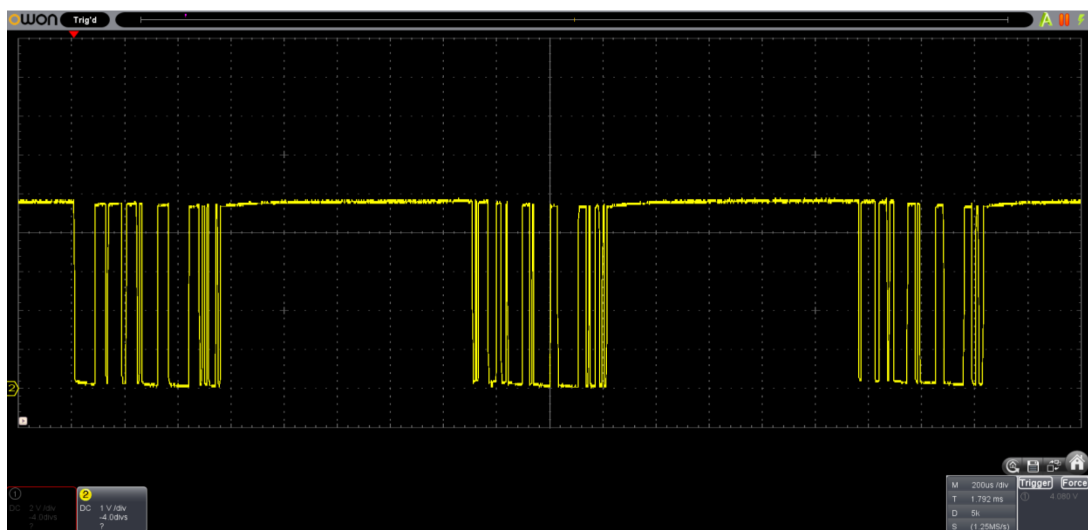
A buszkezelő szoftver ehhez a réteghez köthető. A legkisebb küldési egység (bájtok) kiküldésén és fogadásán kívül minden a bevezető kérdésekkel megfogalmazott elvárások teljesítése a szoftverre hárul.

3.2.1 Csomagkeretezés

De hogy is lesz az elsődlegesen adatfolyam átvitelre használt perifériából csomagátvitelre használható eszköz? Ha meg kellene fogalmazni ebben a kontextusban, hogy mi a csomag, akkor az állományhoz hasonlóan azt az „összefüggő bájtok sorozata”-ként célszerű értelmezni. Így a feladat az, hogy a folyamként érkező adatokat valahogy feldaraboljuk. Több lehetséges módszer is felmerült ennek a megvalósítására: Keretformátum használata, ahol a csomag elejére kódoljuk a csomag hosszát vagy olyan byte kódolási módszert használunk, amivel a csomagok keretezhetőek (ezt később, a PC – busz csatlakozónál felhasználtam). Mindkét módszer esetén a probléma az, hogy – mivel a közeghez egyszerre több eszköz is hozzáférhet és az adást megzavarhatja – elronthatja a keret leíró adatát. Ezzel nem csak a jelenlegi átvitelre szánt csomagot rontja el, de a hosszméret vagy a csomagvég elrontásával, a következő adásban lévő csomagot is, vagy ha méret az eredetinél kisebb vagy szerencsétlen módon a csomag pont úgy sérült meg, hogy az a csomagvég jelzésre hasonlít, az álcsomagok megjelenését eredményezheti. Ebben az esetben valamilyen helyreállítási technikát kellene kidolgozni, ami valószínűleg túlkomplikált működéshez vezethet. Ezzel a problémával a többi buszrendszer valamilyen szinkronizációs vagy modulációs technikával küzd meg: Az SPI a CS (Chips select vonalon keresztül jelzi a csomag kereteit). Az I2C az órajel (SCL) vonalon az átvitel kezdete és befejezése állapottal. A 1-Wire az átvitel végén a reset kiadásával biztosítja, a tranzakciók határát. Az Ethernet

Manchester kódolást használ, aminél az átvitel és az adásszünet jól elkülöníthető. A CAN kódolási technikát használ, minden 5 ugyanolyan bit után egy ellentétes polaritású bitet szúr be, a csomag végét hét egymás domináns logikai szintű átvitt bit jelzi. Ezeknek a módszerek az elő követelménye az, hogy nagyobb kontrollunk legyen az átviteli közeg felett. Vegyük újra szemügyre az UART-ot! A CAN-hez hasonlóan, a hivatalos terminus szerint nem sorolhatjuk a szinkron átviteli módszerek közé, ám a fogadás módja mégis szinkron: start bittel jelzik a csomag kezdetét és a belső időzítőjüket ehhez szinkronizálva mintavételezik a buszt és fogadják a csomagot. Azaz kívülről nézve nem szinkron, de működését tekintve igen. Ehhez hasonló elvet fel lehetne használni a csomagkeretek megállapításához is.[5] A számítógépektől eltérően, a mikrovezérlők esetén a hardverhez közvetlen hozzáférésünk van, így pontosabb adás időzítés megvalósítására és bájt fogadási időpont megállapítására van lehetőségünk.

A végkifejlett az, hogy a csomagokat az időben összefüggő bájtok sorozataként értelmezzük: A csomag kezdetét az első leadott byte jelenti, a csomag végét egy meghatározott adásszünet.



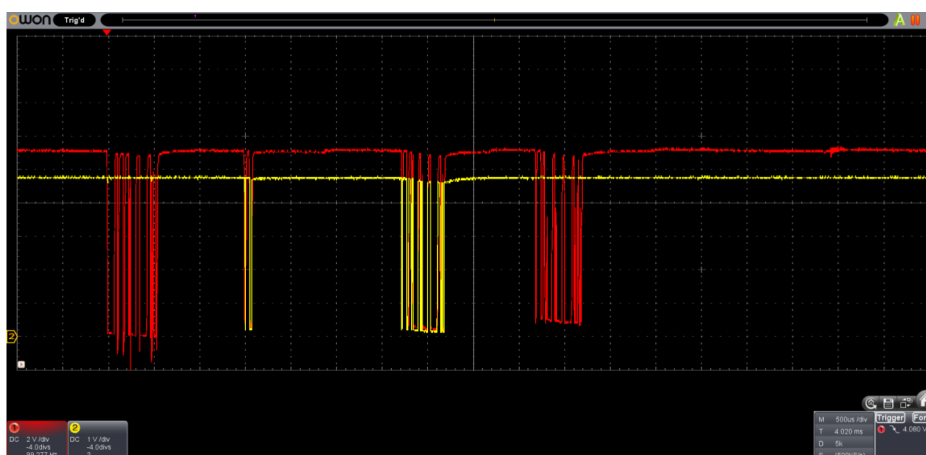
18. ábra Egy broadcast ping és két eszköz válasz csomagjának időfüggvénye, az idő alapú csomag szétválasztás szemléltetésére.

[5] Olyan értelemben hasonlót, hogy külső hatás alapján (start bit – első fogadott bájt ideje) egy belső mechanizmus segítségével (órajel generátor és mintavételező - időzítő) valósítja meg a feladatot (bájt átvitele – csomag átvitele).

3.2.2 Ütközéskezelés

Hogyan észleljük és kezeljük az egyidejű adások ütközését?

Mivel minden egyes kiküldött bájt az összes csatlakoztatott eszközön megjelenik, beleértve az éppen adó eszközt is, így az adó eszköz ellenőrizheti, hogy tényleg az az adat került a hálózatra amit valóban ki kellett küldenie. Azaz adó (TX) regiszterében, a bájt feladása után annak meg kell jelennie a vevő (RX) regiszterében. A módszer az, hogy megjegyezzük a kiküldésre szánt bájt értékét, az feladjuk a TX oldalon. Majd mikor visszaolvassuk (blokkolásos vagy megszakításos úton) ha a kiküldeni szánt érték megegyezik a beérkezettel, akkor az átvitel helyes volt. Ha az értékek eltérnek, akkor valószínűleg más eszköz is épp most kezdte meg az adást. Ekkor a vevő regiszterben a két különböző eszköz által küldésre szánt bájt kombinációja található[6]. Ez az ellenőrzés végigkíséri az egész adási folyamatot, azaz ha a csomag adásának a közepén (ha nem is más eszköz miatt, akár hálózati zaj miatt) hibásan átvitt bájtot állapít meg, ebből ütközésre következtetve, akkor az Ethernethez hasonló módszerrel kezeli azt. Az adást azonnal megszakítja, azaz olyan állapotba áll a buszkezelő szoftver. amiben kivárja a csomag keret végét, továbbá még várakoznak egy véletlenszerűen megválasztott ideig. Azonban ebben az állapotban csomagvételt kezdeményezhet. Az ütközésben nem résztvevő eszközök a „csomag vége" várakozás miatt észlelik az adás befejezését, a csomagot ellenőrzik és hibásnak találják.



19. ábra Ütközés és újraadás szemléltetése két eszközzel, broadcast ping csomag segítségével

[6] Nem feltétlenül az egymáshoz igazított bájtok „vagy" kapcsolatuk található benne, hiszen a két eszköz pár bit eltolással is megkezdheti az adást. Azaz nincs olyan szinkronizációs mód, mint a CAN busz esetén, így a jelenlegi verzióban nem valósítható meg az ütközésfeloldás módszere (CSMA/CR)

A 19. ábra Ütközés és újraadás szemléltetése két eszközzel, broadcast ping csomag segítségével) két buszeszköz ütközése látható. Az egyes csatornán piros színnel a buszvezeték időfüggvénye, a kettes sárgával jelölt csatornán az egyik eszköz TX kimenete látható. Az időfüggvényen négy jól elkülöníthető rész látható.

Az első csomagkeretben a broadcast (üzenetszóró) ping csomag, amelyet minden eszköz fogad és a csomagban lévő parancsot végrehajtja (jelen esetben egy „pong" válasz csomaggal, ezt később részletes tárgyalásra kerül).

A második részben egy ütközést látható. Az időkeretben megfigyelhetjük, hogy a busz vezeték és az eszköz TX lábának időfüggvénye fedésbe van, de az ütközésre csak az adás rövidségéből következtethetünk. Egy címzett csomag minimális hossza minimum 3 bájt[7].

A harmadik ciklusban a kettes csatornán megfigyelt eszköz kezdi meg az adást.

A negyedik ciklusban a másik eszköz forgalmaz.

További fejlesztési lehetőség

Az ütközések észlelése jelenleg nem a leghatékonyabb. Optimális esetben is az ütközést az első bájt vétele után állapíthatjuk meg. Ez azért van, mert a mikrovezérőkben az UART vevő egység úgy van megvalósítva, hogy megszakítást (vagy a státusz regiszterében a vétel befejezését) csak a bájt megérkezése után van módunk észlelni. Más szóval: a start bit észlelésére nincs külön megszakítás vonal vagy státusz regiszter kialakítva. Ez azt is jelenti hogy ütközés esetén a buszon leghamarabb 1 bájt adás ideje plusz a „csomag vége" várakozás leteltével van módunk adás kezdeményezni. A megoldás az lehet, hogy erre felhasználjuk a mikrovezérlő egy olyan lábát, aminek a változásának figyeléséhez (nekünk a lefutó él észlelésére lenne szükségünk) kialakítunk megszakítás vonalat. Ha a busz tétlen állapotban van és jelváltozást észlel, akkor azonnal átállíthatja magát „vétel" módban, ami azt is maga után vonja, hogy már nem kezdeményezhet adást, fogadnia kell a buszról érkező adatot. Ezzel a kiegészítő módszerrel is előfordulhatnak ütközések, de kisebb rájuk az esély a rövidebb észlelési idő miatt. Ebbe a fejlesztésbe olyan szükséges optimalizációk is beletartoznak, mint például hogy a vétel idejére az ütközés észlelő megszakítást le kell tiltani, hogy feleslegesen ne érkezzen megszakítás minden egyes bit változás esetén a buszról.

[7] Ez később a címzésnél kerül kifejtésre, de röviden: 1 bájt célcím plusz 1 bájt forráscím plusz 1 bájt crc8 ellenőrző összeg.

3.2.3 Egyszerűsített elárasztás elleni védelem

Ez a buszrendszereknek egy olyan fontos képessége, ami megakadályozza, hogy egy hibásan megírt vagy beállított eszköz megbénítsa a busz forgalmat azzal, hogy minden lehetséges adási ciklusban adatot küld a hálózatra. Az ütközéskezelés miatt, ha nem is bénítja meg teljesen, de jelentősen elfoglalttá teszi a buszt. Ezt a fejlesztés folyamán megtapasztaltam, így már a jelenlegi verzióba is bekerült egy olyan megoldás, ami újrahasznosítása lehetővé tette az ütközés kezelésének megvalósításánál is[8].

A módszer lényege, hogy az adási ciklus után az adó eszköz extra várakozási időt iktat be, ami alatt viszont megkezdheti egy csomag vételét. A busz ezen állapota a „fairwait" nevet kapta. Ez a módszer csak nagyobb forgalom esetén észlelhető és akkor is az adó eszköznek csak egy adás ciklus utáni időre áll fenn. Az elgondolás az, hogyha az eszköz hozzáfért a buszhoz és azon adatot küldhetett, legyen kedves és adja meg a többi eszköznek is a lehetőséget.

3.2.4 Csomag sértetlenség vizsgálat

A csomag tartalmának helyességét, a közismert CRC ellenőrző összeg mellékelésével ellenőrzöm. Pontosabban a Dallas/Maxim által 8 bitre módosított változatát használom (crc8). A hálózaton lévő eszközök, ha olyan átvitelt használnak, amelyben szeretnének megbizonyosodni a csomagok sértetlenségéről akkor ezt mellékelhetik, illetve vizsgálhatják a csomag végén. A később részletezett host (vagy bootloader) program is így tesz, de az adatkapcsolati réteg szoftvere ezt nem kényszeríti ki. Azaz az ellenőrző összeg használata opcionális. A buszcsatoló program könyvtár nem fűzi minden egyes csomag végéhez hozzá, azaz használata opcionális, de ajánlott az ütközések miatt létrejövő hibás csomagok kiszűréséhez. A később bemutatott PC illesztő esetén is, a csomagot a hálózatra küldő programozónak a csomag végéhez hozzá kell fűznie az ellenőrző összeget, ha olyan

[8] Ebben az állapotban egyéni adási embargót időtartamot adhatunk meg. Ütközés esetén ebbe a „fairwait" állapotba vezéreljük a csatolót és beállítjuk a csomag vége plusz a véletlen várakozási időtartamot.

eszközökhöz szeretné eljuttatni a csomagot amelyek szintén használnak ilyet ellenőrzési módszert.

3.2.5 További fejlesztési lehetőségek

A 3.2.2 előbbvégén említett pontosabb ütközés vizsgáló megoldás megvalósítása és integrálása a jelenlegi beállítási lehetőségekhez.
Mikrovezérlő specifikus programrészek kiszervezése: jelenleg az időzítők inicializálása a buszkezelő könyvtárban található, aminek csak az eszköz független részeket kellene tartalmaznia.
Megszakítás üzemű adási mód megvalósítása: jelenleg a mikrovezérlő az adás idejére teljesen elfoglalt az adatok kiküldésével és a visszaolvasás szinkronizációjával, noha a csomag fogadás esetén sikerült teljes megszakításos üzemmódot megvalósítani.
Módváltás adat tömb és adat szolgáltató függvény között: Jelenleg az adatok kiküldése és fogadása tömbök segítségével történik, amelyek nyilván memóriát foglalnak, minél nagyobb a beállított maximális csomag méret annál többet. Tömbök helyett a könyvtárat használó fejlesztő beregisztrálhatna olyan fogadó és lekérő függvényeket, amelyek feleslegessé teszik a tömbök lefoglalását. Ez azonban más programszervezési formát igényel az alkalmazást fejlesztője részéről, bár ha konfigurálható, azaz opcionális, nem jelent problémát.

3.3 Hálózati réteg (OSI-3)

Támogatja-e a több eszköz vagy eszközcsoport egyidejű címzését (üzenetszórás, broadcasting)? Milyen címzési módszert használ és mennyi a címezhető eszközök felső határa. Kezel-e több közvetetten csatlakozó hálózati réteget (útvonalválasztás, routing), illetve ha igen biztosítja-e a csomagok célba érését és az útválasztási hibák észlelését, javítását (TTL).

3.3.1 Hálózati üzenetszórás és csoportok (broadcast)

Ha áttekintjük az adat kapcsolati réteg leírását, akkor azt az ide vonatkozó következtetést vonhatjuk le, hogy a hálózaton feladott csomagot minden egyes eszköz megkapja. Ez a működési elv fontos az üzenetszórás és eszközcsoportok megvalósításához. A broadcast cím és a csoport címek ábrázolását a „3.3.2 Hálózat címzési módszere” részt tárgyalja, de itt még annyit érdemes megemlíteni, hogy a címzésre előjeles számokat használunk, a 0 a broadcast,- a pozitív címek eszköz,- a negatívak pedig eszköz csoport címek. Az eszközcsoportok ötletét a linux alatt használt folyamat csoportok adták, ami segítségével a csoportban lévő folyamatoknak adhatunk egy paranccsal jelzést. Pl.: kill –HUP $pid, ahol ha a $pid negatív szám, a csoportban lévő összes folyamatnak a HUP jelzést elküldésre kerül.

3.3.2 Hálózat címzési módszere

A címzés tekintetében a tervezés során eldöntik, hogy milyen bit vagy bájt szélességű címeket használjon a rendszer. Minél több bitből áll a cím annál több eszköz címezhető meg a hálózaton, de – mivel minden egyes csomagban megjelenik – adatkeretezési többletként tekinthetünk rá. Ha ezt túl kicsinek választjuk, azzal korlátozzuk a címezhető egységek mennyiségét[9], ha túl nagyra, akkor kisebb adatmennyiségek esetén aránytalanul nagy a csomag mérete a hasznos adathoz képest[10]. Arra is van példa az I2C esetén, hogy lefoglalt címeket felhasználva a címben egy meghatározott bitet beállítva az eredetileg 7 bitre tervezett címzés 11 bitre módosul, így növelhető a címtartomány. Persze ez statikusan, a szabványban meghatározott módon történik.

Ez utóbbi címzési módszer egy rejtett lehetőséget sejtet egy hatékony címzés megvalósítására. Ezt a statikus címbővítési módszert kellene dinamikussá tenni. Mivel az UART Busz esetén a bájt a legkisebb egység ezért ennek dinamikusan bővülő címterét kell megalkotni. Ügyelni kell az eszközcsoportok miatt arra is, hogy negatív számokat is lehessen a címekben ábrázolni. A megvalósításhoz a vezérgondolatot az adta, hogy a címnek tetszőleges

[9] Például: Az IPv6 bevezetése azért vált indokolttá, mert az internet elterjedése és a tendenciájának előrevetítése miatt az IPv4 által biztosított kiosztható címek a jövőben nem lesznek elegendőek.

[10] Például: A minden egyes 1-Wire eszköz 64 bites gyári egyedi azonosítóval rendelkezik, amelyet a Master az eszköz megszólításához használ. Az Ethernet a MAC szinten az átvitelre 48 bites gyári egyedi azonosítókat használ.

hosszúságú bájtok sorozatából kell állnia. Ezek hosszának meghatározásához, hogy ne kelljen a cím előtt még egy mezőt felvinni ami a hosszát határozza meg és ezzel ismét korlátozva a legnagyobb címet, a C stringek[11] mintájára a bájton belül kell a bájt szekvencia végét jelölni.

	7. MSB	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. LSB
Első bájt	E	S	A_5	A_4	A_3	A_2	A_1	A_0
N. bájt	E	A_6	A_5	A_4	A_3	A_2	A_1	A_0

20. ábra Változó hosszúságú címzés bájt szekvencia formátuma

A szekvenciában lévő minden bájt esetén az első „E"-vel jelölt mező (Extend) azt jelzi hogy a cím folytatódik a következő bájton. Ha ez be van állítva (1 az értéke), akkor a következő bájtot is a címmezőhöz tartozóként kezeljük, ha nincs (0 az értéke) akkor az a cím végét jelenti.

Előjeles értékek esetén az első bájtban lévő S mező (Sign), ha be van állítva (1 az értéke) akkor a teljes cím kiolvasása után negatív számként értelmezzük (a pozitív érték előjelét megfordítjuk és kivonunk belőle egyet). Nem előjeles esetén az S mezőbe plusz egy címbit, az A_6 kerül.

Példák az előjeles címekre.

Cím	Első bájt
0	0000_0000
-1	0100_0000
16	0001_0000
-16	0100_1111
-64	0111_1111
63	0011_1111

21. ábra Példa az 1 bájt hosszú címekre

[11] A C programozási nyelv által használt karakterlánc kezelési módszer, aminek a lényege hogy a hasznos szöveget egy speciális ún. termináló karakterrel \0 zárják le. A karakterlánc hossza ilyen módon csak úgy határozható meg, hogy végigmegyünk a láncon egészen eddig a karakterig és megszámoljuk a karaktereket.

Cím	Első bájt	Második bájt
128	1000_0001	0000_0000
-128	1100_0000	0111_1111
8191	1011_1111	0111_1111
-8192	1111_1111	0111_1111

22. ábra Példa a 2 bájt hosszú címekre

Cím	Első bájt	Második bájt	Harmadik bájt
524288	1010_0000	1000_0000	0000_0000
-524289	1110_0000	1000_0000	0000_0000
1048575	1011_1111	1111_1111	0111_1111
-1048576	1111_1111	1111_1111	0111_1111

23. ábra Példa a 3 bájt hosszú címekre

Ezek alapján a n bájt esetén a kiosztható címek száma:

$$A\ n\ = 2^{n*7}$$

Ha a hálózat nem használ előjeles címzést (azaz nem támogatja a csoportokat) akkor a megcímezhető eszközök száma $A\ n\ - 1$. Ha az üzenetszórást se, akkor a 0 cím is kiosztható, így $A\ n$ darab eszköznek osztható cím.

A további tervezés során az elsőjeles címeket használjuk, kiosztható eszközcímek összesen:

$$D\ n\ = \frac{A\ n}{2} - 1$$

A kiosztható csoportok száma:

$$G\ n\ = \frac{A\ n}{2}$$

Ennek a címzési módszernek az az előnye, hogy kevesebb hálózati eszközzel tervezve, azaz telepítés során az eszközöknek kisebb címet adva, a keretezési többlet csökkenthető. Később,

ha több eszközt adnak a hálózathoz, akkor sem szükséges minden egyes egységre új programot feltölteni, ami a nagyobb tartomány címzését lehetővé teszi, egyszerűen csak ki kell osztani az eszköznek egy nagyobb címet. Ettől fogva a megcímezhető eszközöknek csak a mikrovezérlő belső cím típusának a mérete jelent határt. Jelenleg a mikrovezérlőkben int16_t[12] adattípusban kerül a cím tárolásra. Ezt a 3 bájtos formátumból még ki tudják olvasni, de ha az az int16_t tartományán kívül esik, a csomagot eldobják. Viszont ezzel a forrás és célcím esetén is spóroltunk 1-1 bájtot minden egyes átvitt csomagon, ha 63 eszközcímnél többet nem használunk.

Ezek alapján az ajánlott csomag formátum: a csomag elején a célcím és a forrás cím, változó címhosszúsággal kódolva[13], majd a csomag tartalma, végül a lezáró crc8 ellenőrző összeg.

3.4 További rétegek (OSI-4-7)

Azon rétegek összessége, amely az alkalmazás fejlesztőnek egy olyan programozási felületet biztosít, ami elfedi az alacsony szintű működést és így magasabb szintű funkcionalitás valósítható meg a segítségével.

3.4.1 Szállítási és viszony réteg

A szállítási és viszony réteghez olyan funkciók tartoznak, mint például a megbízható adatfolyamok és átvitel létesítése két eszköz között (pl.: TCP). Ez azonban a mikrovezérlők erőforrásaihoz képest, jelen megvalósítási elképzelések szerint túlságosan erőforrás igényes. Ezért egyenlőre ilyen átviteli funkció nem kerül kidolgozásra, megvalósításra. A minimalizált erőforrás igényű megvalósítása további kutatás és fejlesztési feladatot jelent.

[12] 16 bites, előjeles egész szám ami -32768 és +32767 közötti szám ábrázolást tesz lehetővé.

[13] Ezt a sémát használja az Ethernet is a MAC címek esetén, így később optimalizálható a vétel arra, hogy a legelső fogadott cím alapján eldöntse, hogy szükség van-e a csomag további tartalmára, nekünk vagy a csoportunknak szól-e.

3.4.2 Megjelenítési réteg (RPC)

A rendszer jelenlegi kidolgozását tekintve teljesen kötetlen. A csomag formátumnak nincs egyetlen kötött eleme se, csupán egy ajánlás, amit én is felhasználok a végső alkalmazás a projekt könyvtárban[14] „ub_bootloader”-[15]ként megtalálható alkalmazásnál.

Ilyen az RPC réteg is: más projektben bármilyen formátumot felhasználhatunk a csomagokban, de a jelenlegi feladat szerint itt egy olyan általános réteget kell megalkotnunk, ami minimális lenyomatú[16] és könnyen bővíthető a későbbiek folyamán, anélkül hogy a funkcionalitást biztosító könyvtárat módosítani kelljen. Ilyen könyvtárat egyes gyártók biztosítanak, ilyen például az NXP által fejlesztett eRPC[17] aminek azonban jelenlegi ismereteim szerint nem illeszthető csomag alapú rendszerekhez.

A csomag alapra épített RPC rendszer végleges formája a fejlesztés során nyerte el a végleges formáját, azonban az alapötletet a névterek igénye és erre egy megoldási javaslatot az SNMP MIB-ek szolgáltatták. Az SNMP egy olyan protokoll, ami egy számokkal azonosított fa adatstruktúrába rendezett, információt szolgáltató objektumokat tesz elérhetővé. Az egyes objektumokat a számmal azonosított útvonalával érhető el, pl.: 1.3.6.1.2.1.1.1.0 cím alatt a rendszer rövid hardver és szoftver információ kérhető le. Ezeket az útvonalak előre lefoglalt névterek, amelyet az IANA tart karban. Ez a példa jól bemutatja, hogy karakter lánccal azonosított névterek helyett ezt számokkal is megtehetjük.

RPC réteg megvalósítása

Az RPC rendszer 3 központi elemből épül fel: a kérést reprezentáló struktúrából, a diszpécselést végző funkciókból és a funkció láncokból.

[14] A projekt nyílt forráskódú, amely elérhető a github-on: https://github.com/danko-david/uartbus .
A szakdolgozat beadásakor a „tdk” tag-gel ellátott változat került dokumentációra.
[15] A forrás könyvtárban a source/uc/bootaloder/ub_bootloader.cpp állomány.
[16] Kód által és futás közben felhasznált memória minimalizálása
[17] Nyilt forráskódú RPC szerver beágyazott eszközökre: https://github.com/EmbeddedRPC/erpc

```
24 struct rpc_request
25 {
26     int16_t from;
27     int16_t to;
28     uint8_t* payload;
29     uint8_t size;
30     uint8_t procPtr;
31 
32     int16_t (*reply)(struct rpc_request* req, uint8_t args, struct response_part** parts);
33 };
```

24. ábra Az RPC kérést leíró struktúra[18]

A „from” és a „to” a forrás és a célcímet tartalmazza. A „payload” a hasznos adatot, azaz jelenleg a csomagot a címek és ellenőrző összeg nélkül. A „size” a hasznos adat méretét. A procPtr egy indexmutató ami megmutatja, hogy hányadik bájttól kell a következő funkciónak az értelmezést folytatnia. Végül a „reply” egy olyan függvénymutató ami az egyes végfunkciókban meghívhatóak, ezzel lehet a kérést megválaszolni, illetve a választ összeépíteni. Ez utólsóval a rejtett lehetőség az, hogy egy a diszpécselés közbeni funkció kicserélheti ezt a választ biztosító függvényt, ezáltal további funkcionalitást biztosíthat. Pl.: A válaszolásra szánt csomagot elmentheti, ami később egy másik RPC hívásban újra előhívható. Ezzel akár arra is lehetőség nyílik arra, hogy egy speciális névtérben a tranzakciós üzemmód egy részét megvalósítsuk, úgy hogy a „reply” funkciót lecserélve és az utána következő csomagrész a gyökér névtértől újradiszpécseljük. Így ha a válasz csomag valamiért elveszne, azt később kikérhetjük.

```
240 /************************** RPC functions - Bus ******************************/
241 
242 void rpc_bus_ping(struct rpc_request* req)
243 {
244     PORTB ^= 0xff;
245     il_reply(req, 0);
246 }
247 
248 void rpc_bus_replay(struct rpc_request* req)
249 {
250     il_reply_arr(req, req->payload+req->procPtr, req->size - req->procPtr);
251 }
252 
253 void* RPC_FUNCTIONS_BUS[] =
254 {
255     (void*) 2,
256     (void*) rpc_bus_ping,
257     (void*) rpc_bus_replay
258 };
259 
```

25. ábra RPC funkciók és egy névtér függvény lánca[19]

[18] A forrás könyvtárban a source/uc/utils/lib/rpc/rpc.h állomány részlete.
[19] A forrás könyvtárban a source/uc/bootaloder/ub_bootloader.cpp állomány részlete.

Itt megfigyelhető, hogy minden RPC funkció ugyanazzal a szignatúrával rendelkezik. A fenti két funckció (rpc_bus_ping és rpc_bus_replay) a lent látható „RPC_FUNCTIONS_BUS” tömbbe van „beregisztrálva”. Ez valójában egy diszpécser láncként használt tömb, aminek az első (0. indexű) eleme a lánc hossza[20], ezt az egyes RPC funkciók követik.

```
81 /****************************** Dispatch utils ********************************/
82
83 void dispatch_function_chain(void** chain, struct rpc_request* req)
84 {
85     if(0 != (req->size - req->procPtr))
86     {
87         uint8_t cs = arr_size(chain);
88         if(req->payload[req->procPtr] < cs)
89         {
90             void (*func)(struct rpc_request* req)
91             = (void (*)(struct rpc_request* req))
92             chain[req->payload[req->procPtr]+1];
93             if(NULL != func)
94             {
95                 ++req->procPtr;
96                 func(req);
97                 return;
98             }
99         }
100        il_reply(req, 1, ENOSYS);
101        return;
102    }
103    il_reply(req, 1, EBADF);
104 }
```

26. ábra RPC diszpécselést végrehajtó funkció[21]

A 85. soron megvizsgáljuk, hogy van-e még feldolgozható bájt, amin a diszpécselést végrehajthatjuk. A 88. soron ellenőrizzük, hogy a kért funkció létezik-e, ha igen kikérjük a láncból és ha az nem NULL, akkor a „procPtr” indexmutató növeljük hogy a hívott függvény a következő pozícióról folytassa a feldolgozást, majd meghívjuk a függvényt. Hiba esetén az „il_replay” könyvtári funkció segítségével visszajelezzük a hiba okát. Több ilyen láncot egymásba ágyazva az SNMP-hez hasonló számsorral azonosított funkció fa alakítható ki. Mivel a meghívás pillanatában a „procPtr” indexmutató a névteret behatároló rész utánra mutat, a csomag fennmaradó részébe a paraméterek foglalnak helyet. Mivel az erőforrások korlátozottak, a paraméterek típusai nem kerülnek feltüntetésre, így ez nem biztosít a paraméterek helyességének ellenőrzéséhez általános módszert. Válasz csomagban, a csomag kötelező elemein felül, alapértelmezetten a teljes névtér útvonala és a visszatérési érték kerül,

[20] A C karakterláncok mintájára, NULL terminált tömbként is meg lehetett volna valósítani, így viszont támogatja a helyfoglalók használatát azzal, hogy akár a tömb közepén is lehet NULL érték. Bár erre egy „rpc_no_such_function” is jó lehetne abban az esetben.

[21] A forrás könyvtárban a source/uc/utils/lib/rpc/rpc.cpp állomány részlete.

így ha egy eszköz egy időben nem hívja kétszer ugyanazt az eljárást, akkor a hálózaton a híváshoz tartozó válasz azonosítható.

Alkalmazás réteg

Az alkalmazás és a hozzá tartozó infrastruktúra megvalósítása a következő külön fejezetben kerül kifejtésre.

4 Infrastruktúra tervezése

Az infrastruktúra részét a mikrovezérlőben megvalósított szolgáltatások és később a PC-vel összekötött busz, az ahhoz készített könyvtár, ami kezelhető formát ad a buszeszközöknek és legvégül az ennek kezeléséhez készített programok képzik.

4.1 A mikrovezérlőn megvalósított szolgáltatások

Az RPC réteg segítségével a mikrovezérlő által megvalósított funkciókat a hálózaton meghívhatóvá tudjuk tenni. Így egy fontos kérdés maradt: hogyan valósítjuk meg a kódfeltöltést, kód fogadási mechanizmust? A fejlesztés kezdetén egy hálózati bootloader ötlete merült fel, amely ha az eszközt fel szeretnénk programozni újraindul és a bootloaderekhez hasonlóan elkezdi a kód fogadását és a flash memóriában tárolt kód frissítését. A keretrendszernek viszonylag nagy a kódmérete. El kellene kerülni azt, hogy az alkalmazás módban használt kódhoz ezt újra fel kelljen tölteni. A jelenlegi elgondolás szerint az elsődleges program, ami a mikrovezérlőn fut az a bootloader. A cél „alkalmazás” ezen belül egy jól meghatározott kódterületre kerül feltöltésre és a bootloader ezt a programot meghívja. Az ilyen módon való használat esetén a bootloadert inkább „host”, azaz a gazda alkalmazás elnevezés a megfelelőbb. A fejlesztés jelenleg ebben az irányba zajlik. Ám ahhoz hogy ez a kivitel használható legyen a gazda és alkalmazás között kétirányú kapcsolatra van szükség. Ahhoz hogy az alkalmazás be tudja regisztrálni az általa megvalósított funkciókat az RPC névtérbe, a gazda programnak ehhez olyan funkciókat kell biztosítani, ami lehetővé teszi az RPC hívást fogadó funkciók regisztrálását. Valamint a gazda programnak meg kell tudnia

hívni az alkalmazás programot[22]. Mindkét módszer előre definiált programcímekkel került megvalósításra.

```
777 //https://stackoverflow.com/questions/6686675/gcc-macro-expansion-arguments-inside-s
778 #define S(x) #x
779 #define SX(x) S(x)
780
781 __attribute__((noinline)) void call_app()
782 {
783     asm("call " SX(APP_START_ADDRESS));
784 }
785
786 bool has_app()
787 {
788     return 0xff != pgm_read_byte(APP_START_ADDRESS);
789 }
790
791 /********************************** Host tables ******************************/
792
793 //constants, function pointers
794
795 //bus address
796 //app start address [default] = 4096
797
798 void** HOST_FUNCTIONS = (void*[])
799 {
800     (void*) register_packet_dispatch,
801     (void*) may_send_packet,
802     (void*) send_packet,
803     (void*) get_max_packet_size
804 };
805
806 void** HOST_CONSTANTS = (void*[])
807 {
808     (void*) UB_HOST_VERSION,
809     (void*) BUS_ADDRESS,
810     (void*) HOST_TABLE_ADDRESS,
811     (void*) APP_START_ADDRESS,
812 //  (void*) APP_END_ADDRESS,
813     (void*) APP_CHECKSUM
814 };
815
816 //TODO provide mode feature: malloc/free, micros, bus_manage (externalised bus call)
817 //dispatch for the upper namespace over 32
818 void** HOST_TABLE[] =
819 {
820     (void**) HOST_FUNCTIONS,
821     (void**) HOST_CONSTANTS
822 };
823
824 __attribute__((section(".host_table"))) void*** getHostTable()
825 {
826     return HOST_TABLE;
827 }
```

27. ábra Gazda alkalmazás host táblája[23]

[22] Úgy, mint a programok indítása esetén a main() funkciót.
[23] A forrás könyvtárban a source/uc/bootaloder/ub_bootloader.cpp állomány részlete.

A gazda program a 0x0 – 0x1FFF területen helyezkedik el, az alkalmazás program 0x2000 címen felül[24]. A host program jelenleg a 0x1fe0 kezdő címre egy olyan funkciót helyezett el, amivel az úgynevezett host tábla lekérhető. Ez konstansokat, illetve a host által biztosított funkciókat tartalmaz, mint például funkció[25] regisztrálása az alkalmazás névtérbe érkező csomagok fogadásához (register_packet_dispatch), küldhető-e a buszra adat (may_send_packet), illetve csomag küldése a hálózatra (send_packet). Ez a tömb bővíthető, például mivel a malloc/free használatára is szükség volt az új programkód fogadásához. Azért hogy az alkalmazás kódban ez ne kerüljön duplikálásra, a gazda program ezt a funkciót is szolgáltathatja.

A másik irány, a gazda programból az alkalmazás meghívása. Az alkalmazás is egy jól ismert memória helyen található. Az alkalmazás programok készítéséhez készítettem egy minimális keretet, ami az arduinóhoz hasonlóan a setup és a loop funkciók felülírásával inicializálhatják magukat és a loop funkcióban ciklikus feladatokat kezelhetnek.

```
 2 #include "ub_app_wrapper.h"
 3
 4 //https://stackoverflow.com/questions/6686675/gcc-macro-expansion-arguments-inside-string
 5 #define S(x) #x
 6 #define SX(x) S(x)
 7
 8 __attribute__((noinline)) void*** getHostTableAddress()
 9 {
10     asm("jmp " SX(HOST_TABLE_ADDRESS));
11 }
12
13 __attribute__ ((weak)) void setup(){};
14
15 __attribute__ ((weak)) void loop(){};
16
17 int main(){}
18
19 #include <avr/io.h>
20
21 static volatile bool initialized = false;
22
23 //app_start section starts at 0x2000
24 __attribute__((noinline, section(".app_start"))) void ub_app()
25 {
26     if(!initialized)
27     {
28         init_ub_app();
29         setup();
30         initialized = true;
31     }
32     loop();
33     asm ("ret");
34 }
```

28. ábra UARTBus alkalmazás keret forrása[26]

---

[24] Ez függ a gazda program fordítási méretétől. A jelenlegi méret ATmega328 céleszközre, AVR-GCC 4.9.2 fordító felhasználásával lett meghatározva. A kód mérete a beadás időpontjában 0x1a04 bájt azaz kb. 6,6 Kb.

[25] Olyan kezelő függvény, amit a gazda program hív meg ha az alkalmazás számára új csomag érkezett, így valósítva meg a csomagok átadását.

[26] A forrás könyvtárban a source/uc/utils/ub_app/ub_app_wrapper.cpp állomány részlete.

A 0x2000 címen az ub_app funkció helyezkedik el. Ez inicializálja a keret által biztosított funkciókat, majd az alkalmazás kódot a setup meghívásán keresztül. Ezt a funkciót a host minden ciklusban meghívja. Az alkalmazás kód futtatása után a host ellenőrzi, hogy érkezett-e új csomag és azt kell-e diszpécselni. Ez a ciklus fut a hostba.

Ezek az információk a RPC funkciók segítségével lekérhetőek. Fontos hogy ezek a memória címek nincsenek kőbe vésve, a jövőben változhatnak. A fordításhoz ezeket a címeket le kell kérni a céleszköztől és így az eszköznek megfelelő futtatható kódot kell belőle fordítani.

További fejlesztési lehetőségek

A szolgáltatások megvalósítása koránt sem teljes, pl.: az eszközcsaládok lekérése nincs megvalósítva, ehhez jobban át kell tekinteni a többi lehetséges céleszköz, hogy azokon az eszközazonosító lekérése hogyan valósítható meg, végső esetben ezt szoftveres úton, fordítási időben is megadható. Továbbá az RPC névterek rendeltetésének pontosabb meghatározására és átrendezésére lehet szükség, a jelenlegi szervezési módja kezdetleges. Egyes buszkezeléssel kapcsolatos paramétereket beállíthatóvá kellene tenni, mint pl. az eszköz címe. Továbbá olyan funkciók, mint a „válaszoljon az az eszköz, akinek az egyedi azonosítója X", amelyek hasznosak lehetnek egyes eszközök áttelepítésénél és új címek kiosztásánál anélkül, hogy az eszköz host kódját újra kelljen programozni. Olyan a programozási környezetet érintő feladatok is elvégzésre várnak, mint a hostra beérkező megszakítások továbbítása az alkalmazásnak.

4.1.1 Felhasználói funkciók névtere

Az alkalmazás funkciók elérésének teszteléséhez a tárolóba mellékeltem egy teszt alkalmazást, amivel a PORTB-re csatlakoztatott LED ki és bekapcsolható.

```
 8 /********************************* App section ********************************/
 9
10 void software_reset()
11 {
12     //wdt_enable(WDTO_15MS);
13     //MCUSR = ~8;
14     //while(1){}
15     asm ("jmp 0x00");
16 }
17
18 void packet_received(struct rpc_request* req)
19 {
20     uint8_t* data = req->payload + req->procPtr;
21     uint8_t ep = req->size - req->procPtr;
22     if(0 == ep)
23     {
24         return;
25     }
26
27     if(0 == data[0])
28     {
29         PORTB = 0x0;
30     }
31     else if(1 == data[0])
32     {
33         PORTB = 0xff;
34     }
35     else if(2 == data[0])
36     {
37         software_reset();
38     }
39 }
40
41 void setup()
42 {
43     register_packet_dispatch(packet_received);
44 }
```

29. ábra Egyszerű tesztalkalmazás[27]

Az itt látható (29. ábra Egyszerű tesztalkalmazás 43.sorában) a „register_packet_dispatch”, a host táblában található függvénynek az alkalmazásba importált változata. Ennek a függvények a meghívása esetén a hívás a host kódterületére ugrik és az átadott „packet_received” függvény címét a hostban a 30. ábra A host RPC gyökér diszpécsere) a 631. soron látható változóba tárolja el. Ezzel a kapcsolat megvalósult. Ha egy új csomagot amelynek 2 bájt hasznos tartalma: [32, 1], akkor a hostban a 30. ábrán látható „dispatch” funkcióba 622. sorban az első bájt beolvasásra kerül, amelynek értéke 32 és ezek alapján a 630. és 631. sorokon lévő kód ágában lévő részlet kerül meghívásra. Itt a következő bájtra állítjuk a kérésben lévő „procPtr” indexmutatót és az „app_dispatch” függvénymutatóval jelzett függvényt meghívjuk. Ez ebben az esetben a hívást a 29. ábra Egyszerű tesztalkalmazás látható) „packet_received” funkcióba irányítja (amit az előbb beregisztráltunk). Itt az RPC kérés nyers adatait tartalmazó „payload” tömböt, az indexmutató segítésével eltoljuk, így a nulladik index a tömbe lévő „1”-es decimális értékre mutat. Ez a 33. sorban azt eredményezi

[27] A forrás könyvtárban a source/uc/utils/apps/ub/test.cpp állomány részlete.

hogy a PORTB-re kötött LED bekapcsol. Ugyanezen hívás mentén, ha a [32, 0] adatot küldünk az eszköznek, akkor a PORTB-re kötött LED kikapcsol. A [32, 2] hívás esetén a „software_reset” kerül meghívásra, ami az eszköz újraindulását eredményezi.

```
615 void dispatch(struct rpc_request* req)
616 {
617     if(0 == req->size)
618     {
619         return;
620     }
621
622     int ns = req->payload[req->procPtr];
623
624     if(req->from >= 0 && 0 < ns && ns < 32)
625     {
626         dispatch_root(req);
627     }
628     else if(NULL != app_dispatch && 32 == ns)
629     {
630         ++req->procPtr;
631         app_dispatch(req);
632     }
633     //rewrite app dispatch ot dispatch ns >= 32
634     else if(ns > 32)//these are also user space channels, dispatch them.
635     {
636     }
637 }
```

30. ábra A host RPC gyökér diszpécsere[28]

4.2 Számítógép csatolása a buszrendszerbe

Az UARTBus-ra kapcsolt mikrovezérlők alapvetően primitív feladatokat látnak el. Összetettebb funkciók megvalósítása, ami több eszköz koordinációját igényli nehéz egy mikrovezérlőben megvalósítani. Ezért szükség van egy olyan megoldásra, amivel bonyolult feladatok megvalósítására alkalmas eszközt is, mint például egy számítógép csatlakoztani lehet a buszhoz. A számítógépen futó program, amely a buszrendszerre csatlakoztatott eszközöknek parancsot küld és onnan adatokat kér le, lehet akár egy asztali alkalmazás vagy webszerver webes alkalmazással. Jelenleg CLI[29] alkalmazások vannak a buszrendszer diagnosztizálásához kifejlesztve, ami később bemutatásra kerül.

[28] A forrás könyvtárban a source/uc/bootaloder/ub_bootloader.cpp állomány részlete.
[29] Command Line Interface: Parancssoros felhasználói felület.

4.2.1 Számítógép buszcsatoló

A számítógép csatolásához egy olyan mikrovezérlő lett felhasználva, amely legalább 2 UART porttal rendelkezik. Az egyik az UARTBus könyvtárat felhasználva a buszrendszerhez csatlakozik. Illetve a másik vonal a számítógéphez van csatlakoztatva. A buszról érkezett csomagokat a számítógép felé soros vonalon elküldi, illetve a küldeni kívánt csomagokat fogadja a számítógéptől és azt elküldi a buszra. A számítógép közvetlenül csatlakoztatása a buszra nem megoldható, mivel az azon futó operációs rendszer miatt nincs kontrollunk az időzítések felett. Real-time operációs rendszer esetén megvalósítható, ha az UART periféria és az időzítés felett megvan az a kontroll mint a mikrovezérlők esetében, de ez a technika általános megoldást próbál nyújtani a számítógépek illesztésére. Itt a számítógép oldaláról ismét előtérbe kerül a csomag keretezése amelyről a korábban szó esett. Itt a 3.2.1 „Csomagkeretezés" fejezetnél említett bájt kódolás módszert használom fel, amivel a csomagkeretek megállapíthatóak. Az UARTBus esetén a lehetséges ütközések miatt nem volt célszerű a használata, viszont a számítógép és a busz csatoló közötti pont-pont kommunikáció esetén nem fordulhat elő ütközés, ezért ez a technika megbízhatóan használható. A technika alapját a programozási nyelvekben használt „escape character" (feloldójel) módszer biztosította: Például C nyelven a karakter láncokat felső idéző jelek között vihetjük be. Például: "string". Ha ilyen idéző jelet szeretnénk bevinni, akkor az egy \ feloldó jellel együtt kell bevinnünk: pl.: "quotation: \"lorem ipsum\"". Ezzel a módszerrel több új speciális karaktert bevihetünk, avagy a feloldó jel mögött bármilyen más a csatornán használt karakter állhat.

```
76 void relaySerial(uint8_t* data, uint8_t size)
77 {
78     uint8_t e = 0;
79 
80     for(uint8_t i=0;i<size;++i)
81     {
82         if(data[i] == PACKET_ESCAPE)
83         {
84             encode[e++] = PACKET_ESCAPE;
85         }
86         encode[e++] = data[i];
87     }
88 
89     encode[e++] = PACKET_ESCAPE;
90     encode[e++] = ~PACKET_ESCAPE;
91 
92     flashLed();
93     SERIAL_PC.write((uint8_t*) encode, (uint8_t) e);
94 }
```

31. ábra Arduino segítségével megvalósított csomagtovábbító a számítógép felé[30]

[30] A forrás könyvtárban a source/uc/utils/apps/arduino/bus_connector.cpp állomány részlete.

A buszról érkező csomagok minden féle csomagformátum nélkül, nyersen ebben a funkcióba érkezik meg, amely feloldó jelekkel látja el a csomagot és továbbítja a számítógépre csatolt UART portra.

A csomagok vétele esetén a számítógép felől érkező folyamot kell csomagokra feldarabolni.

```
108 void readSerial()
109 {
110     while(SERIAL_PC.available())
111     {
112         byte b = SERIAL_PC.read();
113 
114         if(mayCut)
115         {
116             if(b == (byte)~PACKET_ESCAPE)
117             {
118                 flashLed();
119                 arduino_ub_send_packet(&BUS, decode, ep);
120                 ep = 0;
121             }
122             else
123             {
124                 decode[ep++] = b;
125             }
126             mayCut = false;
127         }
128         else
129         {
130             if(b == (byte)PACKET_ESCAPE)
131             {
132                 mayCut = true;
133             }
134             else
135             {
136                 decode[ep++] = b;
137             }
138         }
139     }
140 }
```

32. ábra Arduino segítségével megvalósított csomagdaraboló a számítógép felöl érkező adatokhoz[31]

A számítógépen futó program esetén ennek a működésbeli pontos mását meg kell alkotni ahhoz, hogy az adatfolyam feldarabolása működhessen. A PACKET_ESCAPE fordítási időben meghatározható érték, a buszillesztőnek és a számítóképes programnak ugyanazt az értéket kell használnia. Választása azért megfontolandó, mivel minden egyes adat bájt, ami megegyezik a feloldó karakterrel 2 bájtot eredményez az adatfolyamban. Ez alapján érdemes

[31] A forrás könyvtárban a source/uc/utils/apps/arduino/bus_connector.cpp állomány részlete.

olyan értéket választani, ami ritkábban fordul elő a csomagokban. A buszillesztő jelenleg ATmega2560-hoz készült.

Forráskódja a tárolóban a source/uc/utils/apps/arduino/bus_connector.cpp fájlban található, ami mellett található egy „target_atmega2560_make_and_upload.sh” nevű futtatható script, ami az említett mikrovezérlőhöz lefordítja a scriptben megadott 115200 baud ráta használatához a forrást és feltölti a script paraméterében megadott USB-soros porton, amihez a feltöltő egy bootloaderrel ellátott és csatlakoztatott mikrovezérlőt feltételez. A busz csatoló feltöltése után, mikrovezérlő számítógéphez csatlakoztatott soros portján a csatolt UARTBus rendszerrel kapcsolatot teremthetünk.

A 3.4.2 „Megjelenítési réteg (RPC)” részben említettem egy fontos részletet, hogy a csomagkeretnek nincs kötött formátuma. Ezen a részen figyelhető meg, hogy a számítógép felől érkező csomag módosítás nélkül feladásra kerül. Azaz az a nyers adattömb amit a folyamból beolvasunk változatlan formában kerül elküldésre, anélkül hogy címzést vagy ellenőrző összeget mellékelnénk a csomaghoz. Ez az általánosított megvalósítás tökéletes terep a kísérletezésre: több csomag formátumot is kipróbálhatunk anélkül, hogy a busz csatoló kódját módosítanunk kellene. Ez azt is jelenti, hogy a címzésről és az ellenőrző összeg mellékeléséről az alkalmazásnak kell gondoskodnia. Valójában a busz csatlakozónak nincs saját címe és minden csomagot vesz az UARTBus-ról és továbbít a számítógép felé. Ezt úgy is mondhatjuk, hogy ez az eszköz az OSI-2-es szinten működik. Ezért mikor csomagot továbbítunk az UARTBus-ra csatoló segítségével, akkor a „csomagot szúrunk be a hálózatba”[32] kifejezést használom, illetve „a buszon forgalmazott csomagok megfigyelését”.

4.2.2 RPC szerver

A busz csatlakozóval a soros porton keresztüli csatlakozás elérhetővé tette a buszrendszert, de ehhez a porthoz egy időben csak egy alkalmazás csatlakozhat, ami korlátozza a beavatkozási lehetőséget. Például ha egy vezérlő alkalmazás fut és lefoglalja a soros portot, akkor egy esetleges üzemzavar esetén ezen a porton nem tudunk még egy, a hibakeresésre használt

[32] A küldés azt a mechanizmust sejteti, hogy a feladott adatcsomagot valamilyen eljárás megcímzi, mellékli a feladót, ellenőrző összeget és elküldi a hálózatra. Beszúrás esetén nincs ilyen eljárás, csak az összefüggő bájt csoport kiküldése a hálózatra.

alkalmazással becsatlakozni. Ezért egy olyan elosztó számítógépes alkalmazásra van szükségünk, ami lehetőséget nyújt egyszerre több szoftvernek is becsatlakozásra.

Ehhez készítettem java programozási nyelven egy olyan RPC szervert[33] alkalmazást, ami erre a soros portra csatlakozik és egy TCP porton hallgatózva, az oda érkező kéréseket az UARTBus-nak továbbítja, illetve funkcióhívás segítségével a buszon forgalmazott csomagok lekérhetőek.

```
1  package eu.javaexperience.electronic.uartbus.rpc;
2
3⊕ import java.io.Closeable;
5
6  public interface UartbusConnection extends Closeable
7  {
8      //sending data
9      public void sendPacket(byte[] data) throws IOException;
10
11     public String getAttribute(String key) throws IOException;
12     public void setAttribute(String key, String value) throws IOException;
13
14     public long getCurrentPacketIndex() throws IOException;
15     public byte[] getPacket(long index) throws IOException;
16
17     public byte[] getNextPacket() throws IOException;
18 }
```

33. ábra A számítógépes réteg által használt interface az UARTBus-on történő adatcseréhez[34]

A szerver oldalon ennek az interface[35]-nek a megvalósítása található, ami soros porton keresztül a busz csatolóhoz kapcsolódik. Az RPC szerver egy nyelv agnosztikus[36] csatlakozási felületet biztosít. Ehhez a szerverhez csatlakozhatnak azok az alkalmazások, amelyek a buszon lévő eszközökkel szeretnének kommunikálni vagy csak onnan adatot gyűjteni. Számunkra a „sendPacket" funkció, mely az átadott bájt tömböt beszúrja a hálózatra és a „getNextPacket" funkció lényeges. Ez utóbbi az UARTBus-ról beérkezett egy csomaggal tér vissza. Egy fontos megvalósítási részlet, hogy a „getNextPacket" a megvalósításban puffereket hasnál, azaz nem hagy ki csomagokat a hívás között. Ezt azt jelenti, hogy ha az első hívás után megkapjuk az első csomagot a hálózatról és tételezzük fel, hogy várakozunk egy másodpercet az újabb „getNextPacket" hívással, ami alatt a buszon 3 csomag forgalmazása lezajlott, akkor a következő „getNextPacket" hívásnál a második csomagot kapjuk vissza és nem az ötödiket. Azaz az esetleges hívás várakoztatásával nem maradunk le

[33] Ez teljesen független az UARTBus-nál ismertetett RPC-től, ez a számítógépes rendszer szoftveres környezetében biztosít funkcióhívási felületet.

[34] A forrás könyvtárban a source/java/uartbus/src/main/java/eu/javaexperience/electronic/uartbus/rpc/UartbusConnection.java állomány teljes tartalma

[35] Az „interface" itt és továbbiakban, azt az objektum orientált programozási nyelvekben használt programozás technikai elemet jelenti, ami egy objektum által (a használat oldaláról) biztosított és (a megvalósítás oldaláról) a megvalósításra előírt funkciókat tartalmazza szignatúrájukkal együtt. A nem OOP nyelvenkben (pl.: C) ezt a szerepet a header állományok töltik be, azonban OOP nyelvek esetén ez egy jól meghatározott objektum egység keretein belül.

[36] Programozási nyelv független, azaz nem kizárólag a szolgáltatás írott nyelvén (java), hanem bármely nyelven (javascript, PHP, C) megírt program is csatlakoztatható.

a két hívás között forgalmazott csomagokról. Az alkalmazásban is ezek a funkciók érhetőek el, ám az „UartbusConnection” interface megvalósítása mögött ekkor nem a soros port áll, hanem egy TCP kapcsolat, ami az UARTBus RPC szervernek az továbbítja. Ilyen módon az alkalmazások dinamikusan becsatlakozhatnak a busz kommunikációba.

4.3 CLI eszköztár

Így olyan alkalmazásokat készíthetünk, amely a 4.2.2 (RPC szerver) fejezetben ismertetett módon dinamikusan be tudnak csatlakozni a buszrendszerbe.

Ezek a programokat egy konzolból meghívható indító csomagba gyűjtöttem össze, ami a projekt könyvtárban található „scripts/assemble_uartbus.sh” script segítségével fordítható le. Ugyanebben a mappában a „scripts/add_command_ub.sh” instrukciót ad arra, hogy hogyan tegyük linux alatt az „ub” parancsot elérhetővé a parancsértelmezőnk számára.

Ha a programot lefordítjuk az „ub” parancsot kiadjuk akkor egy – ugyan kezdetleges formában – de az elérhető alkalmazásokat kilistázza a program.

```
$ ub
public interface Cli
{
        public void console(String[] a);
        public void ihex(String[] a);
        public void compile(String[] a);
        public void upload(String[] a);
        public void rpcServer(String[] a);
        public void ping(String[] a);
        public void packetloss(String[] a);
        public void collisionPacketloss(String[] a);
}
$ 
```

34. ábra Cli eszközalkalmazások listázása

Az „ub $parancsnév” kiadásával az egyes programokhoz ír segítséget a gyűjtő alkalmazás.

Az „ihex” program a feltölthető intel HEX állományok ellenőrzésére használható, ellenőrzi, hogy minden bejegyzés sértetlen-e, majd az ellenőrzés végén kiírja a program valós méretét[37]. A „compile” még befejezetlen, a buszon lévő eszközre való fordítást valósít meg, ez később kombinálásra kerül az „upload” paraccsal ami egy lefordított intel HEX alkalmazást ellenőriz

[37] Ez utóbbira a linux alatt elérhető „size” parancs is alkalmas.

és tölt fel a buszon lévő eszközre. Az „rpcServer” segítségével a 4.2.2 (RPC szerver) fejezetben említett szerver indítható el. Alapértelmezetten a 2112 porton várja a TCP kapcsolatokat, hacsak a „-p” kapcsolóval másképp nem rendelkezünk, a „-b” és a „-s” segítségével a soros port megadása kötelező az indításhoz. Indítása pl.: „ub rpcServer -b 115200 -s /dev/ttyUSB0”. A további eszközök részletesebben leírásra kerülnek.

4.3.1 CLI konzol

Az UARTBus konzol segítségével a buszon folyó forgalmat figyelhetjük meg és szúrhatunk be csomagokat. Paraméter nélküli hívás esetén alapértelmezetten a localhost-on a 2112 portra próbál csatlakozni ahol alapértelmezetten az RPC szerver is fut.

```
$ ub console
[2019-04-08 11:56:51.146U]      <6163/1>
ed
>63
From address: 63
<1
To address: 1
1:0
63:1:1:0:189
1:0
63:1:1:0:189
<1
To address: 1
1:0
63:1:1:0:189
64:1:0:2:2:75
```

35. ábra UARTBus konzol

A „>” karakterrel és a számmal a „küldő” állítható be, ezt a feladó címet tesszük a hálózatra beszúrandó csomaghoz. Azért esett a 63-ra a feladó címre a választás, mert ez a 3.3.2 (Hálózat címzési módszere) fejezetben ismertetett címzés szerint ez a legnagyobb cím, ami elfér 1 bájtban.

A „<” karakter segítségével a célcímet határozhatjuk meg, amire a begépelt csomagot küldjük. Az általunk begépelt csomagok pontos összeállítása nem látható a konzolon, a válasz csomagok viszont teljességében, azaz a nyers címeket és a csomag végén az ellenőrző összegek is megjelennek. A címek beállítása után az „1:0” kiadásával, ezt a bájt szekvencia értéket címzéssel és ellenőrző összeg keretezésével elküldi a hálózatra. A jelenlegi

kialakításban ez az jelenti, hogy feladunk egy csomagot amit a 63-as eszköz küld[38] az 1-es eszköznek és mivel RPC funkció diszpécselést használunk az „1:0" csomag tartalom azt eredményezi hogy az 1-es névtérben lévő 0 indexű funkció meghívásra kerül. A hívásunk a 25. ábra RPC funkciók és egy névtér függvény lánca) kódrészen kerül kiszolgálásra. Az 25. ábra RPC funkciók és egy névtér függvény lánca, a 253. sorban lévő RPC_FUNCTIONS_BUS lánc az 1-es névtérhez van csatolva és a hívás az ebben a 0 indexű funkcióhoz kerül, ami ugyanezen ábra 242. sorában látható „rpc_bus_ping" híváshoz kerül és a benne lévő funkció válaszolja meg a ping kérést, amit végeredményként a konzolon látunk. Mivel bármilyen csomagot beszúrhatunk a hálózatra, diagnosztikára is jól használható.

```
$ ub console
[2019-04-08 12:22:43.859U]
ed
>63
From address: 63
<0
To address: 0
1:0
!63:0
63:1:1:0:189
63:2:1:0:89
```

36. ábra UARTBus konzol, eszközök felfedezése

Például a 63-as címről az üzenetszóró (broadcast) címre küldhetünk egy ping csomagot, aminek az eredménye az, hogy minden eszköz feldolgozza a kérést és válaszol a csomagra. Ilyen módon, a válaszokból megállapítható hogy milyen eszközök vannak jelen a hálózaton. A 36. ábra UARTBus konzol, eszközök felfedezése) válaszok során egy ütközést is megfigyelhetünk. Azok a válaszcsomagok amelyeknek az ellenőrző összege helytelen felkiáltójellel kezdődnek. Az ütközés feloldása után mindkét buszeszköz csomagját láthatjuk a konzolon.

4.3.2 CLI ping

Egy kiválasztott eszköz folyamatos pingelését teszi lehetővé, a linux ping parancsához hasonlóan. Az „-f" kapcsolóval a feladó címét, a „-t" kapcsolóval a cél eszközt adhatjuk meg,

[38] Ez az eszköz nem létezik a hálózaton, csupán a „nevében" küldjük ki.

az „-i” a ping csomagok kiküldése között idő intervallum milliszekundum pontosságú beállítására használható, a „-c” a kiküldeni szánt csomagok mennyiségére, ha ez nincs megadva a pingelés folyamatos.

```
$ ub ping -f 63 -t 1 -i 800 -c 10
[2019-04-08 12:43:39.574U]      <21387/1>
ed
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
ping 63 > 1
63:1:1:0:189
```

37. ábra UARTBus CLI ping

Ez a program akkor hasznos, hogyha egy eszköz kommunikációjával rövid idő alatt megismétlődő probléma áll fenn. A példával ellentétben nem minden csomagot követ válasz pong csomag, illetve a csomagok meghibásodhatnak, de ezzel az eszközzel ez nyomonkövethető.

4.3.3 CLI csomagveszteség mérés - packetloss

A ping parancsnál egy kifinomultabb eszköz, amivel a csatoló és az eszköz közötti csomag veszteséget tudjuk mérni. Az „-f”, „-t”, „-c” és „-i” a 4.3.2 (CLI ping) programhoz hasonló, a további „-l” kapcsolóval azt az idő intervallumot állíthatjuk be milliszekundum pontossággal, hogy mikor nyílvánítjuk a kiküldött csomagot elveszettnek. A program végén vagy annak megszakítása esetén (linux terminálon a Ctrl+C kiadása) kiírja a csomagveszteségi statisztikát.

```
$ ub packetloss -f 63 -t 1 -l 200 -c 100 -i 100
[2019-04-08 12:51:45.273U]      <8808/1>        {JavaExpe
ed
0. pong
1. pong
2. -- loss --
3. pong
4. pong
5. pong
6. pong
7. pong
8. pong
9. pong

90. pong
91. pong
92. pong
93. pong
94. pong
95. pong
96. pong
97. pong
98. pong
99. pong
Statistic:
        Sent packets: 100
        Received packets: 98
        Lost packets: 2
        Packet loss ratio: 2 %
```

38. ábra UARTBus CLI packetloss eszköz kimenete

4.3.4 CLI ütközéses csomagveszteség mérés – collisionPacketloss

A csomagveszteség mérést és az eszközök felfedezésének ötvözésével olyan diagnosztikai eszköz fejleszthető, amivel a buszon lévő összes eszköz csomagvesztesége mérhető versenyhelyzetben. A programkapcsolói a 4.3.2 (CLI ping) programhoz hasonlóak, azt leszámítva hogy a célcím „-t” nem adható meg mivel egy broadcast címre adja fel a program azokat a csomagokat amivel a mérést végzi.

```
$ ub collisionPacketloss -f 63 -i 100 -c 100
[2019-04-08 13:03:25.153U]      <4348/1>
ed
0. broadcast ping
1. broadcast ping
2. broadcast ping
3. broadcast ping
4. broadcast ping
5. broadcast ping
6. broadcast ping
7. broadcast ping

95. broadcast ping
96. broadcast ping
97. broadcast ping
98. broadcast ping
99. broadcast ping
Last turn, waiting a bit for the last packets.
Statistic for bus device 1:     Sent packets: 100       Received packets: 89    Lost packets: 11        Packet loss ratio: 11 %
Statistic for bus device 2:     Sent packets: 100       Received packets: 92    Lost packets: 8 Packet loss ratio: 8 %
```

39. ábra UARTBus CLI collisionPacketloss - csomagveszteség mérése üzenetszóró pingeléssel

A futás végén eszközönkénti statisztikába összegzi a program hogy az egyes eszközök felé milyen a csomagvesztési ráta. Az oka különböző lehet pl.: buszeszköz annyira elfoglalt hogy

több csomag érkezik hozzá, mint amennyit fel tud dolgozni, vagy a busz csatolója részlegesen meghibásodott így nem tudja az adást venni vagy küldeni. Akárhogy is legyen, ez a program rámutathat a problémás busz eszközökre vagy eszközcsoportra.

5 Összefoglalás

A buszrendszerek kutatásával sikerült annyi ismeretet összegyűjtenem, amely képessé tett arra, hogy egy új buszrendszert alkothassak meg, egy eredetileg nem erre a célra szánt, de a legtöbb mikrovezérlőben megtalálható periféria segítségével. A jelenlegi szakdolgozat témája azonban közel sem tekinthető befejezettnek. Az egyes fejezetek végén ismertetett fejlesztési lehetőségek több továbbhaladási irányt is kijelölnek. Az egyik legfontosabb szempont viszont, hogy az utolsó fejezetben ismertetett eszközök segítségével a kijelölt fejlesztések lehetőségek hatásossága mérhetővé vált. A rendszer funkciójának demonstrációjához szükséges lenne olyan a mikrovezérlőben szánt próba vagy demóalkalmazásoknak, amelyek részletesebben bemutatja a rendszer képességeit. Ez utóbbira az egyes részfeladatok és a dokumentálási feladat volumenének alábecslése miatt nem jutott elég idő.

Irodalomjegyzék

[1] Könyv: Linux programozás. Asztalos Márk, Bányász Gábor, Levendovszky Tihamér. 2012, SZAK kiadó. ISBN 978-963-9863-29-3

[2] Könyv: Számítógép Hálózatok. Andrew S. Tanenbaum, David J. Wetherall. 2013, Panem kiadó. ISBN 978-9-635455-29-4

[3] Könyv: Hatékony Java. Joshua Bloch 2008, Kiskapu kiadó. ISBN 978-963-9637-50-4

[4] Weboldal, angol Wikipédia https://en.wikipedia.org/wiki/OSI_model Link utoljára ellenőrizve: 2019.03.26

[5] Weboldal, angol Wikipédia https://en.wikipedia.org/wiki/Carrier-sense_multiple_access_with_collision_avoidance Link utoljára ellenőrizve: 2019.04.03

[6] Weboldal, angol Wikipédia https://en.wikipedia.org/wiki/IEC_61334 Link utoljára ellenőrizve: 2019.04.03

[7] PDF dokumentáció http://www.sti.uniurb.it/romanell/Domotica_e_Edifici_Intelligenti/110504-Lez10a-KNX-Architecture%20v3.0.pdf Link utoljára ellenőrizve: 2019.04.03

[8] PDF, dokumentáció (alkatrész adatlap) https://www.nxp.com/docs/en/data-sheet/P82B96.pdf Link utoljára ellenőrizve: 2019.03.28

[9] Weboldal, angol Wikipédia https://en.wikipedia.org/wiki/KNX_(standard) Link utoljára ellenőrizve: 2019.04.03

[10] Weboldal https://www.maximintegrated.com/en/app-notes/index.mvp/id/148 Link utoljára ellenőrizve: 2019.03.29

[11] PDF, kutatási publikáció https://www.idosi.org/mejsr/mejsr23(ssps)15/64.pdf Link utoljára ellenőrizve: 2019.03.29

[12] Weboldal (Fórum) https://electronics.stackexchange.com/questions/430025/ethernet-connection-using-only-one-twisted-pair/ Link utoljára ellenőrizve: 2019.04.01

[13] Weboldal, német Wikipédia

https://de.wikipedia.org/wiki/Carrier_Sense_Multiple_Access/Collision_Resolution

Link utoljára ellenőrizve: 2019.03.28

[14] PDF, alkatrész adatlap (TJA1055T)

https://www.nxp.com/docs/en/application-note/AH0801.pdf

Link utoljára ellenőrizve: 2019.03.28

[16] PDF, alkatrész adatlap (ATMega168)

http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-9365-Automotive-Microcontrollers-ATmega88-ATmega168_Datasheet.pdf

Link utoljára ellenőrizve: 2019.04.06

[15] PDF, alkatrész adatlap (ATMega328)

https://www.sparkfun.com/datasheets/Components/SMD/ATMega328.pdf

Link utoljára ellenőrizve: 2019.04.06

[16] PDF, alkatrész adatlap (STM32F103x8)

https://www.st.com/resource/en/datasheet/cd00161566.pdf

Link utoljára ellenőrizve: 2019.04.06

[17] PDF, alkatrész adatlap (PIC18F2550)

https://ww1.microchip.com/downloads/en/devicedoc/39632c.pdf

Link utoljára ellenőrizve: 2019.04.06

[18] Weboldal, angol Wikipédia

https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

Link utoljára ellenőrizve: 2019.04.06

[19] Weboldal, információs oldal

http://www.oid-info.com/get/1.3.6.1.2.1.1.1

Link utoljára ellenőrizve: 2019.04.06

[20] Weboldal, magyar Wikipédia

https://hu.wikipedia.org/wiki/Intel_HEX

Link utoljára ellenőrizve: 2019.04.06

Ábrajegyzék

1. ábra KNX csomagkeret formátum [7] 4

2. ábra KNX busz szegmensei 5

3. ábra SPI egy Master és 3 Slave konfiguráció. 6

4. ábra SPI átvitel szemléltetése 6

5. ábra I2C busz csatlakozás egyszerűsített kapcsolási rajza 7

6. ábra I2C egy adatátviteli ciklusának időfüggvénye 8

7. ábra 1-wire adatátvitel időfüggvénye 9

8. ábra Ethernet 10BASE2, T és termináló BNC csatlakozó 10

9. ábra CAN adatátvitel időfüggvénye 12

10. ábra UART adatátvitel szemléltetése 13

11. ábra UART közösített átvitel tesztje 17

12. ábra Mikrovezérlő csatlakoztatás a buszra 19

13. ábra Ohmos buszillesztő 19

14. ábra Ohmos buszillesztő időfüggvénye 115 kbps sebességen 21

15. ábra N-MOSFET buszillesztő 22

16. ábra MOSFET buszillesztő időfüggvénye 115 kbps sebességen 23

17. ábra Busz feszültségének időfüggvénye MOSFET illesztővel 500 kbps sebességen 24

18. ábra Egy broadcast ping és két eszköz válasz csomagjának időfüggvénye, az idő alapú csomag szétválasztás szemléltetésére. 26

19. ábra Ütközés és újraadás szemléltetése két eszközzel, broadcast ping csomag segítségével 27

20. ábra Változó hosszúságú címzés bájt szekvencia formátuma 32

21. ábra Példa az 1 bájt hosszú címekre 32

22. ábra Példa a 2 bájt hosszú címekre 33

23. ábra Példa a 3 bájt hosszú címekre 33

24. ábra Az RPC kérést leíró struktúra 36

25. ábra RPC funkciók és egy névtér függvény lánca 36

26. ábra RPC diszpécselést végrehajtó funkció 37

27. ábra Gazda alkalmazás host táblája 39

28. ábra UARTBus alkalmazás keret forrása 40

29. ábra Egyszerű tesztalkalmazás 42

30. ábra A host RPC gyökér diszpécsere ... 43
31. ábra Arduino segítségével megvalósított csomagtovábbító a számítógép felé ... 44
32. ábra Arduino segítségével megvalósított csomagdaraboló a számítógép felől érkező adatokhoz ... 45
33. ábra A számítógépes réteg által használt interface az UARTBus-on történő adatcseréhez 47
34. ábra Cli eszközalkalmazások listázása ... 48
35. ábra UARTBus konzol ... 49
36. ábra UARTBus konzol, eszközök felfedezése ... 50
37. ábra UARTBus CLI ping ... 51
38. ábra UARTBus CLI packetloss eszköz kimenete ... 52
39. ábra UARTBus CLI collisionPacketloss - csomagveszteség mérése üzenetszóró pingeléssel ... 52

Ábra források

---

[i] http://www.sti.uniurb.it/romanell/Domotica_e_Edifici_Intelligenti/110504-Lez10a-KNX-Architecture%20v3.0.pdf 12. oldal ábrája

[ii] http://www.sti.uniurb.it/romanell/Domotica_e_Edifici_Intelligenti/110504-Lez10a-KNX-Architecture%20v3.0.pdf 10. oldal ábrája

[iii] https://upload.wikimedia.org/wikipedia/commons/f/fc/SPI_three_slaves.svg

[iv] https://upload.wikimedia.org/wikipedia/commons/b/bb/SPI_8-bit_circular_transfer.svg

[v] https://www.hobbielektronika.hu/cikkek/kommunikacio_alapjai_-_soros_adatatvitel.html?pg=5
[vi] https://upload.wikimedia.org/wikipedia/commons/6/64/I2C_data_transfer.svg
[vii] https://upload.wikimedia.org/wikipedia/commons/1/12/1-Wire-Protocol.png
[viii] https://upload.wikimedia.org/wikipedia/commons/5/5f/BNC-Technik.jpg
[ix] https://upload.wikimedia.org/wikipedia/commons/5/5e/CAN-Bus-frame_in_base_format_without_stuffbits.svg
[x] https://upload.wikimedia.org/wikipedia/commons/f/f3/RS-232_timing.svg