Autóipari kommunikációs hálózatok vizsgálata

Laboratóriumi mérések a
***Beágyazott és ambiens rendszerek laboratórium* tárgyhoz**

V5h

Csak belső használatra!

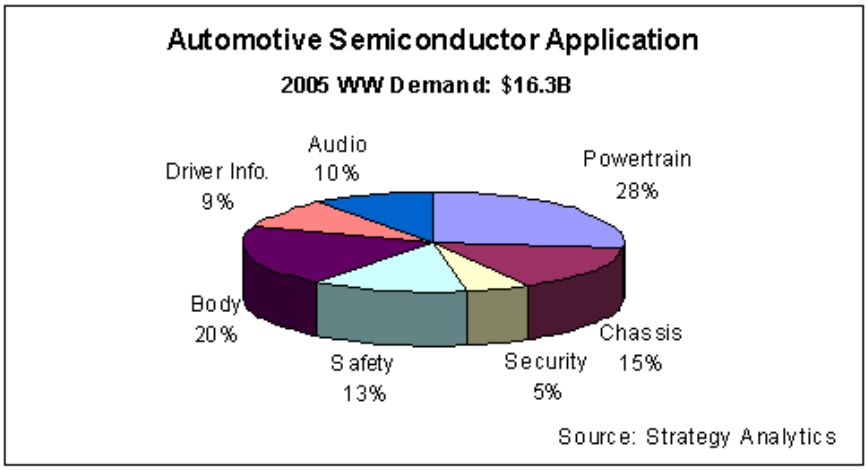
Scherer Balázs, dr. Tóth Csaba
Bosch – Beágyazott Rendszerek Laboratórium
BME MIT

2008-2016.

TARTALOMJEGYZÉK

1. Bevezetés

Egy mai felsőkategóriás autóban 40-80 ECU (Electronic Control Unit) található, és piaci felmérések szerint az autóipari fejlesztések 90%-a kötődik valamilyen módon az elektronikához. Az autókban található ECU-kat funkciójuk alapján általában a következő nagyobb csoportokba sorolják: powertrain systems (erőátviteli rendszerek, pl. motorvezérlő, váltóvezérlő), chassis systems (váz vagy „kaszni" rendszer, pl. fékek, sebességszenzorok), body systems (utastér-elektonika, pl. ablakemelő, világítás, központi zár), multimedia systems (multimédia rendszer, pl. autórádió, hangosítás). Vannak akik, külön csoportba sorolják a biztonsági rendszereket (safety, pl. légzsák, ABS), a védelmi rendszereket (security, pl. ugrókódos ajtónyitó, indításgátló) és a vezetői információs rendszereket (driver info, pl. GPS, tolatóradar). E rendszerek egy 2005-ös felmérésen alapuló százalékos arányát az 1.1 ábrán láthatjuk.

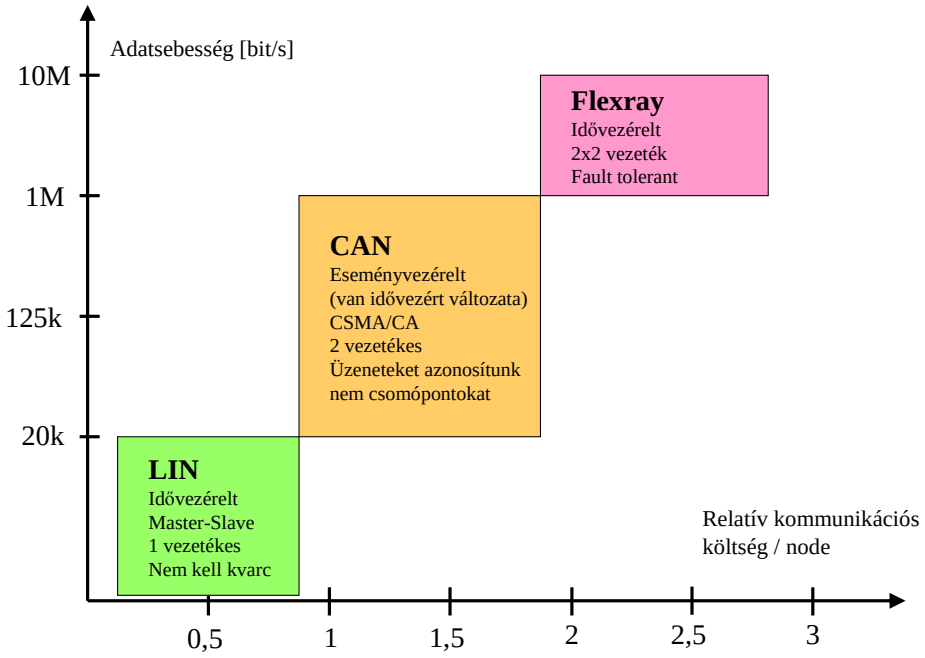


1.1 ábra. Autóipari elektronikus rendszerek aránya

Az egyre több ECU és az egyes alrendszerek közötti együttműködés szükségessége fokozatosan kikényszerítette, hogy a kezdetben különálló egységként létező ECU-k egyre több társukkal lépjenek kapcsolatba, mind komplexebb kommunikációs hálózatot hozva létre. Egy mai autóban 3-5 független kommunikációs hálózat működik. Ezek a kommunikációs hálózatok általában nem egyetlen fizikai kialakítást és protokollt követnek, hanem az adott alkalmazási kör számára legjobb megoldást alkalmazzák. A manapság legelterjedtebb három kommunikációs protokoll a CAN (Controller Area Network), a LIN (Local Interconnect Network) és a FlexRay. Ennek a három kommunikációs protokollnak az adatsebesség és a relatív kommunikációs költség alapú összehasonlítását az 1.2 ábrán figyelhetjük meg.

Az 1.2 ábrán jól látható, hogy a LIN, CAN és a FlexRay is más és más követelményű hálózatokat céloz meg.

Az 1980-as évek elején kifejlesztett CAN a mai napig a legtöbbet alkalmazott autóipari kommunikációs technológia. A nagysebességű (általában 500 kbit/s) CAN hálózatokat leggyakrabban a powertrain és chassis rendszerekben használják, de vannak olyan járművek, ahol a fékvezérlés is dedikált, nagysebességű CAN kommunikáción keresztül történik. Az alacsony (pl. 125 kbit/sec) sebességű CAN hálózatok fő felhasználási területe az utastér-elektronika (body systems).



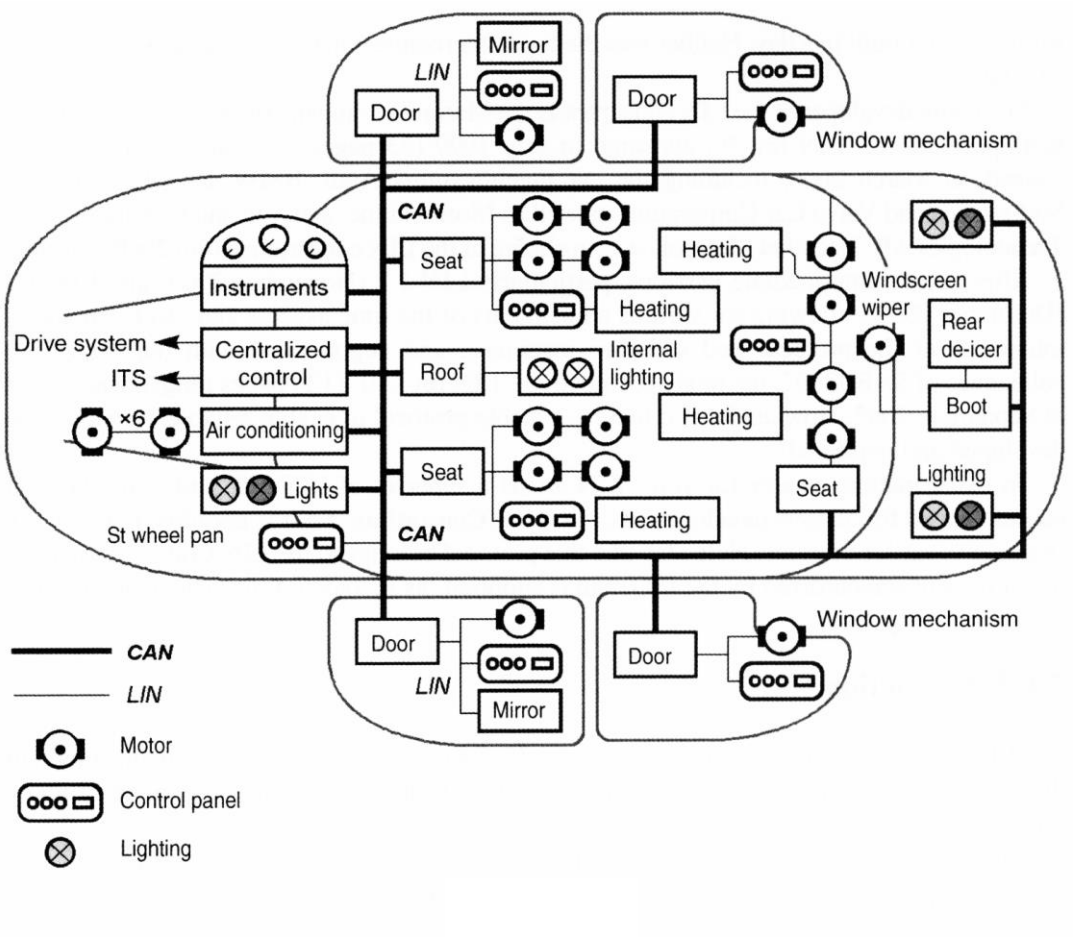
1.2 ábra. Autóipari hálózatok összehasonlítása

Az 1990-es évek végén kifejlesztett, a CAN-nél lényegesen egyszerűbb és olcsóbb LIN fő célja az volt, hogy a CAN kiegészítő hálózataként a legkisebb részegységekig is eljutassa a digitális vezérlést és kommunikációt. LIN hálózatra tipikus példa, amikor egy ajtó összes elektronikus részegysége (mint pl. ablakemelő, központi zár, visszapillantótükör-mozgatás) és ezek kezelőfelületei alkotnak egy LIN hálózatot, amely egy gateway (átjáró) segítségével csatlakozik a body rendszer CAN hálózatához.

A 2007-ben szabványosított FlexRay célja a CAN képességeit meghaladó időkritikus és nagysebességű és nagy-megbízhatóságú alkalmazások lefedése. Felhasználása elsősorban a biztonságkritikus területeken és a drive-by-wire rendszerekben várható, de valószínűleg egyes, eddig CAN busszal megoldott powertrain funkciók migrációja is felfedezhető lesz a FlexRay irányába.

A CAN és LIN hálózatoknak az autón belüli együttélésére jó példa az 1.3-as ábrán bemutatott body rendszer.

Az ábrán jól látható, hogy a LIN alhálózatok egy nagyobb sebességű CAN gerinchálózatra csatlakoznak (az adott elrendezés egy képzeletbeli példa). Szintén megfigyelhető az ábrán *Centralized control* névvel ellátott blokk, amely az összes rendszerrel kapcsolatban áll, és egy központi csomópont szerepét tölti be. A szervizben egy megfelelő műszerrel ennek a központi gateway-nek a diagnosztikai portjára csatlakozva az összes alrendszer működését ellenőrizhetjük.

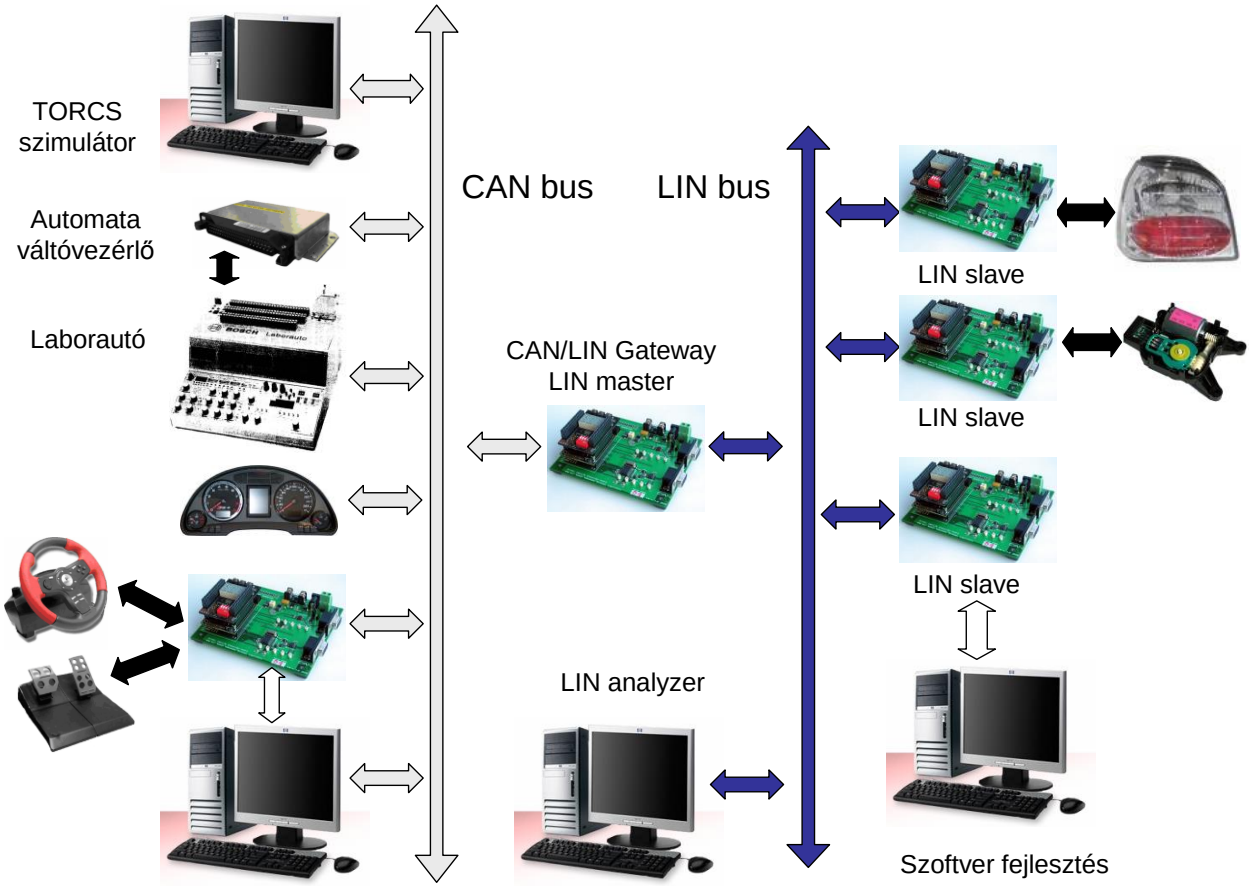


1.3 ábra. Egy minta body rendszer

2. A Bosch Labor mérési összeállítása

Az autóipari mérések során a CAN és LIN hálózatokat tanulmányozzuk. A Bosch Beágyazott Rendszerek Laboratóriumban (IE225) a 2.1 ábrán látható virtuális autón fogjuk végrehajtani a méréseket.

A labor-összeállítás két szeparált részből áll: a CAN méréshez tartozó, bal oldalon látható részből és a jobb oldali, a LIN méréshez tartozó hálózatból. Az egyes mérésekhez tartozó összeállítást a 2.1-es és 2.2-es alfejezet mutatja be. Az ott bemutatott perifériákon kívül mindkét mérőrendszerhez tartozik protokollanalizátor szoftver, mitmót fejlesztőkörnyezet és oszcilloszkóp is.



2.1 ábra CAN/LIN labor-összeállítás

2.1 CAN mérési elrendezés

A CAN mérés feladatait egy részben szimulált, részben valóságos autós perifériákból és készülékekből álló rendszeren végezzük. A mérési összeállítás tartalmaz autókban használatos beágyazott rendszereket, illetve az autó egyes részeit PC-vel vagy speciális célhardverrel szimuláló modulokat. A rendszer összes eleme CAN-buszon keresztül össze van kötve, ugyanakkor szinte mindegyik modul rendelkezik saját analóg jelekkel is, amelyek szintén összeköttetésben lehetnek a többi modullal (ezeket az összeköttetéseket az ábrán nem jeleztük, egyrészt a jobb áttekinthetőség miatt, másrészt mert a mérés szemszögéből ezek nem fontosak).

A vezetői felület

A vezetői felületen keresztül tudjuk az autóvezetést szimulálni. Ennek a modulnak az a feladata, hogy gerjesztéseket adjon az autómodell számára (például fék- és gázpedál állapota). A mérésben a vezetői felületet vagy a Laborautó vagy egy videojátékhoz készült gáz- és fékpedál, valamint egy kormány fogja megjeleníteni (természetesen, ez utóbbi nem teljesen valóságos, hiszen ez egy drive-by-wire rendszer lenne, amelyet a normál autókban biztonsági okokból egyelőre nem alkalmazzák).

Műszerfal

Ez egy valóságos, a Bosch által szériában gyártott periféria. Kilométerórát, fordulatszámmérőt, üzemanyagszint-jelzőt és a drágább autókra jellemző egyéb kijelzőket tartalmaz. A jelzések egy része CAN-buszon keresztül vezérelhető.

Váltóvezérlő

A váltóvezérlő egység szintén egy valóságos autós periféria (Bosch sériatermék VW és Audi gépkocsikban használt modell). Ez a készülék egy meglehetősen komplex periféria: rengeteg analóg bemenettel és néhány digitális kimenettel rendelkezik, továbbá a helyes működéséhez CAN üzenetekben kapott paraméterekre is szüksége van (motorfordulatszám, gázpedál állása, motornyomaték stb.).

Laborautó

A Laborautót a Bosch cég a váltóvezérlői fejlesztésére, tesztelésére fejlesztette ki, amely gyakorlatilag teljes szimulációt tartalmaz az autó többi részéről. A Laborautó potenciométerei és más kezelőszervei segítségével beállítható az autó szinte valamennyi paramétere az alapjárattól kezdve a pillangószelep állásán keresztül egészen az olajnyomásig.

TORCS autószimulátor

A TORCS program szimulálja a virtuális autónk mozgását. Ehhez felhasználja a Laborautótól kapott motor adatokat, a váltóvezérlőtől kapott váltóállás információkat, valamint a vezetői parancsokat, majd a teljes rendszer számára visszajelzi az autónk pillanatnyi sebességét.

2.2 LIN mérési elrendezés

A LIN mérés feladataiban több szimulált elemet fogunk felhasználni, mint a CAN mérésben. A mérések nagyobb része fog kötődni egyszerűbb részegységek programozásához, amire a gyári ECU-kon nem lenne lehetőségünk. A LIN hálózat a következő elemekből áll:

CAN/LIN gateway és LIN master

Ez a hálózat legfontosabb eleme. A *CAN/LIN gateway és LIN master* teremt összeköttetést az alacsony szintű LIN alhálózat és a komplett CAN gerinchálózat között (1.3 ábra). A mérési elrendezésünkben a *CAN/LIN gateway és LIN master* szerepe, hogy ütemezze a LIN buszra csatlakozó perifériák működését, illetve, hogy kapcsolatot tartson a CAN mérésben használt virtuális autóval, ezáltal biztosítva a LIN alrendszer működését befolyásoló külső információkat. Ilyen külső információ például a fékpedál állapota egy féklámpa vezérlésében, az autó sebessége a központi zár esetében stb. A *CAN/LIN gateway és LIN master* fizikai megvalósítása a 2.1 ábrán látható *mitmót* alapú fejlesztőkártya.

LIN slave-ek

A *mitmót* alapú LIN slave-ek kezelik az egyszerű autós perifériáinkat, mint az ablakemelőt és központi zárat szimuláló motorokat és a hátsó lámpákat. A valóságban a LIN slave-ek lényegesebben egyszerűbb hardverrel is rendelkezhetnének, hiszen a legtöbb esetben az egész elektronika beleférne a vezérelt perifériába, ahogy az a 2.2 ábrán is látható. A laborban az egyszerű fejleszthetőség miatt ugyanazokat a mitmót alapú fejlesztőkártyákat használjuk a CAN node, a LIN master és a LIN slave megvalósításra.

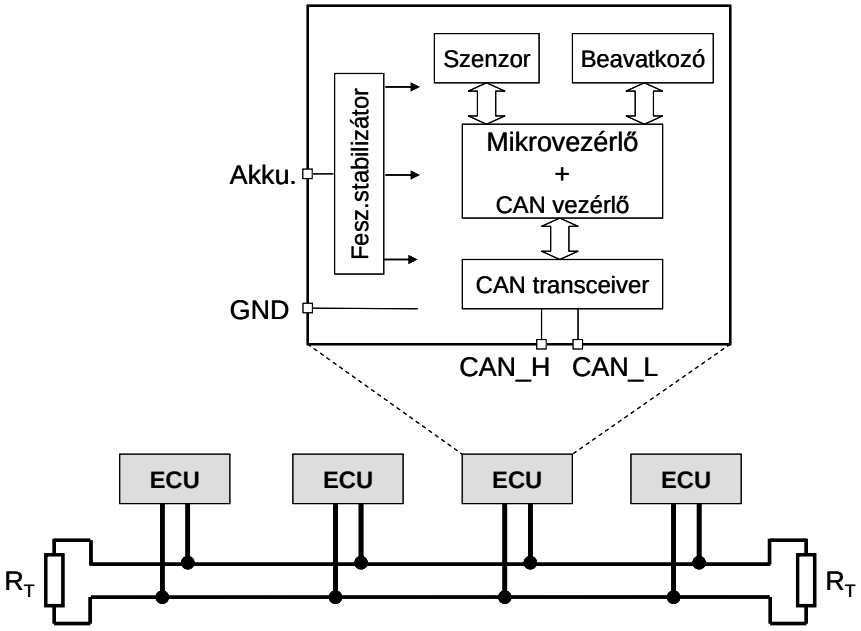
2.2 ábra. Egy LIN slave egység közvetlenül a beavatkozóba integrálva (illusztráció)

3. CAN: Controller Area Network

A Robert Bosch cég által az 1980-as évek elején kifejlesztett CAN-t mind a mai napig széles körben használják. A sokmilliónyi CAN hálózatnak kb. egyharmadát építették be autókba, a többit orvosdiagnosztikai készülékekben (röntgen, CT), automatákban, ipari gyártóberendezésekben használják. A CAN nemzetközi szabvány (ISO 11898).

A CAN a helyi hálózatok (LAN-ok) egy speciális fajtájához, a field-buszokhoz (ipari buszokhoz) tartozik. A field-buszok legtöbbjére jellemző, hogy csak a legszükségesebb OSI rétegeket valósítják meg: a fizikai réteget, az adatkapcsolati réteget (esetleg ennek csak a közeg-hozzáférési alrétegét) és az alkalmazási réteget; a többi réteg hiányzik. A CAN architektúrához csak az alsó másfél réteg tartozik, amelyet kiegészítenek valamilyen alkalmazási réteggel, amely nem része a CAN protokollnak (magasabb rétegként, Higher Layer-ként hivatkoznak rá). A CAN fontosabb jellemzői:

- Busz topológia (szórásos típusú hálózat: a hálózatra adott keretet mindenki veszi)
- Tetszőleges topográfia (általában busz, pont-pont vagy csillag)
- Többszörös hozzáférés (CSMA), nem destruktív ütközéskezelés (huzalozott ÉS kapcsolat)
- Egycímes keretformátum (a címnek inkább adatazonosító és prioritást meghatározó szerepe van)
- 1 Mbit/s maximális adatátviteli sebesség (jellemzően 125 és 500 kbit/s)
- Az áthidalható maximális távolság 40 – 500 m (sebességtől függően)
- Többféle adatátviteli közeg, legtöbbször csavart érpár
- Non-Return To Zero (NRZ) bitkódolás, a transzparens átvitelt bitbeszúrással, bitkiejtéssel biztosítják (bit-stuffing)
- Rövid, változó hosszúságú keretek (0-8 byte hosszú adatmező)



3.1 ábra. CAN-buszra csatlakozó elektronikus egységek

Egy beágyazott rendszerekből álló CAN hálózat felépítését mutatja a 3.1 ábra. Az elektronikus vezérlő egységek (*Electronic Control Unit, ECU*) itt csavart érpáras CAN-buszra csatlakoznak. A busz mindkét vége hullámimpedanciával le van zárva (120-120 ohm).

CAN keretformátumok

A CAN négyféle keretformátumot definiál:

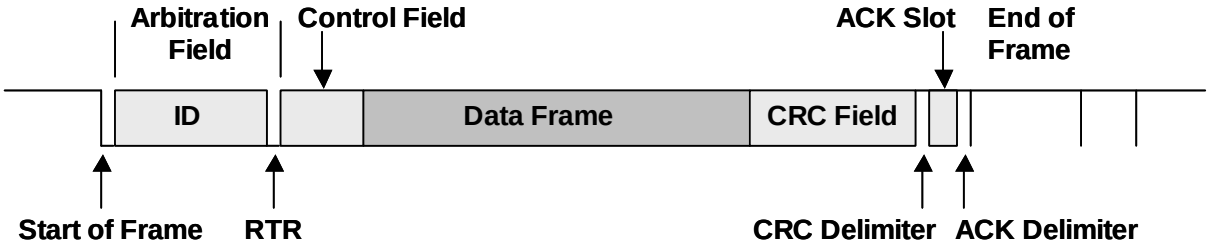
- Adatkeret (Data Frame)
- Hibakeret (Error Frame)
- Távoli keret (Remote Frame)
- Túlcsorduláskeret (Overload Frame)

Számunkra csak az első két keret érdekes, ezért az alábbiakban ezeket ismertetjük.

Adatkeret (Data Frame)

Az adatkeret tartalmazza az alkalmazások számára hasznos információt. Röviden így foglalhatnánk össze a feladatát: „Halló, itt van az X azonosítójú adat, aki akarja, használja fel.” A CAN 2.0A vagy „standard CAN” változat adatkeretének formátuma a 3.2 ábrán látható. Az adatkeret mezőit az alábbi táblázat foglalja össze.

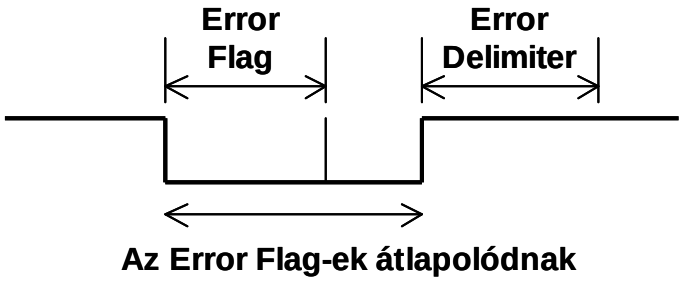
Név	Hossz	Érték	Leírás
Start of Frame	1 bit	0	A keret kezdetét jelzi.
ID (Identifier) Field*	11 bit		Többnyire az adatmező tartalmát azonosítja. Az RTR bittel együtt meghatározza a keret prioritását is. (A CAN 2.0B vagy „Extended CAN” változat 18 bittel kibővíti az ID mezőt.)
RTR (Remote Transmit Request)	1 bit	0	Távoli adáskérésnél RTR=1, egyébként RTR=0.
Control Field - r1, r0 - Data Length Code	 2 bit 4 bit	 0 0 xxx x	Az adatmező hosszát adja meg byte-okban.
Data Field	0-64 bit		Max. 8 byte hosszúságú adatmező. A tartalma tetszőleges lehet.
CRC Field	15 bit		Ellenőrző összeg, a Start of Frame bittől a CRC végéig terjed a hatóköre.
CRC Delimiter	1 bit	1	
ACK Slot	1 bit	1/0	A keretet hibátlanul vevők ezt a bitet nullába állítják (Tx=1/Rx=0).
ACK Delimiter	1 bit	1	
End of Frame	7 bit	111 111 1	A keret végét jelzi.
Intermision	3 bit	111	Keretek közötti szünet.



3.2 ábra. CAN adatkeret (Data Frame)

Hibakeret (Error Frame)

A keret funkcióját röviden így lehetne összefoglalni: „Vigyázzatok, ez egy hibás keret!” Ha az adó adás közben vagy bármelyik vevő vétel közben hibát észlel, kiadja ezt a figyelmeztető keretet. Az adó később újraadja a keretet. A hibakeretet az Error Flag és az Error Delimiter alkotja. Az Error Flag 6 db azonos értékű bitből áll (Error Active módban 6 db 0, Error Passive módban 6 db 1), az Error Delimiter pedig 8 db 1 értékű bitből. A később ismertetendő bitbeszúrási technika miatt normális adatforgalomban 6 azonos értékű bit nem követheti egymást, az Error Flag így egyértelműn felismerhető. A bitbeszúrási szabály megsértésére a többi állomás is hibakeretet ad ki, így az Error Flagek átlapolódnak.



3.3 ábra. CAN hibakeret (Error Frame)

Arbitráció és prioritás

A CAN-es arbitráció azt használja ki, hogy az állomások huzalozott ÉS kapcsolatban vannak egymással. Ha valamennyi adó 1 értéket ad ki (recessive, elengedett érték), akkor a buszon is egyes érték jelenik meg. Ha bármelyik állomás lehúzza a buszt (domináns, meghúzott vagy 0 érték), akkor a buszon nulla érték jelenik meg.

Egy állomás akkor kezdhet el adni, ha szabad a busz, egyébként meg kell várnia, hogy a busz felszabaduljon. Ha egyszerre több állomás kezd el adni, akkor az adóik az első lefutó élre összeszinkronozódnak, és bitről bitre egyszerre hajtják meg a buszt. Adás közben az adó folyamatosan veszi is a jelet a buszról, és összehasonlítja az adott és a vett biteket. Itt kap szerepet a huzalozott ÉS kapcsolat. Ha a kiadott bit és a vett bit nem egyezik meg, akkor az adó feltételezi, hogy ütközés történt, és abbahagyja az adást, majd egy későbbi időpontban megpróbálja újraadni a keretet. Az az állomás, amelyik ugyanazt veszi, mint amit kiad, zavartalanul folytatja az adást. Az arbitrációs versenyben az az állomás nyer (marad adásban), amelyiknek az arbitrációs mezőjében előbb szerepel nullás bit (arbitrációs mező=ID+RTR bit), vagyis amelyiknek kisebb érékű az azonosítója. Az ID értéke egyúttal az adás sorrendjét, prioritását is meghatározza.

Ez az arbitráció nem destruktív (szemben pl. az Ethernettel), ugyanis valamelyik állomás mindenképpen elkezdhet adni, tehát nem történik idő- és sávszélességvesztés. Az arbitrációs versenyből kieső állomások egy későbbi időpontban újra versenyezhetnek a busz használatáért.

CAN ID

A CAN keretek ID mezőjének nem feltétlenül azonos a szokásos számítógép-hálózatok címmezőjével. Például a LAN-okban a cím egy adott állomást (csomópontot) azonosít, ezzel szemben a CAN hálózatokban az ID mező bármit azonosíthat. A CAN protokoll nem mondja meg, hogy mi a jelentése az ID-nek, ezt a magasabb szintű protokollok határozzák meg. Az ID hossza, a bitek adási sorrendje rögzített, de a tartalmára vonatkozóan nincs előírás. Az ID legtöbbször az adatmezőben elküldött adat azonosítója (neve). Az autók esetében a különböző paraméterekhez (változókhoz) rendelnek azonosítókat, pl. egyedi ID-je van a motor fordulatszámának, a jármű sebességének stb. A vevők az ID vizsgálatával döntik el, hogy fogadják-e (továbbítsák-e a felsőbb réteg felé) az adott paramétert tartalmazó keretet vagy dobják el.

A CAN fizikai rétege

A CAN NRZ kódolást használ bitbeszúrással. A bitbeszúrás gondoskodik arról, hogy elegendően gyakran legyen jelátmenet a buszon (ez a bitszinkronizációt könnyíti meg).

A huzalozott ÉS kapcsolat megvalósításához kétféle jelállapot szükséges: az egyik egy recessive (elengedett) állapot, a másik egy domináns (meghúzott) állapot. Az előbbihez 1, az utóbbihoz 0 bitérték tartozik.

A CAN-hez többféle fizikai réteg használható. Leggyakrabban az ISO 11898-2, ún. nagysebességű (high speed, max. 1 Mbit/s) és az ISO 11898-3, ún. kissebességű (low speed, max. 125 kbit/s) fizikai réteget használják. Mindkettő csavart érpárat alkalmaz szimmetrikus adóval és szimmetrikus vevővel (hasonlítanak az RS-485-re). A kissebességű változat hibatűrő képességekkel is rendelkezik. A különböző változatok nem (feltétlenül) tudnak együttműködni, ezért fokozottan figyelni kell az egyes interfészek specifikációjára.

A CAN interfészeknek nemcsak a maximális sebessége limitált, hanem gyakran alsó sebességhatárt is megadnak, pl. egyes meghajtók nem tudnak 10 vagy 50 kbit/s sebesség alá menni.

A kábel hossza az alkalmazott sebességtől függ. Alapvetően az arbitráció szab határt, mivel a kibocsátott jelnek jóval egy bitidőn belül oda-vissza be kell tudnia futni a teljes kábelt. Ha az interfész optocsatolót is tartalmaz (amelynek jelentős lehet a késleltetése), akkor a távolság tovább csökken. Néhány tájékoztató érték a távolságra és a hozzá tartozó sebességre:

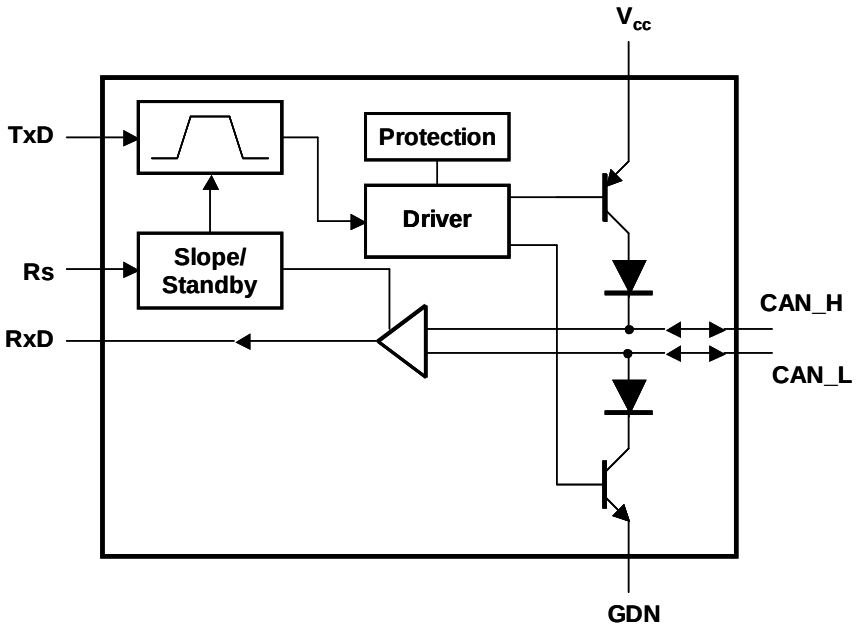
- 100 m (330 ft), 500 kbit/s
- 200 m (650 ft), 250 kbit/s
- 500 m (1600 ft), 125 kbit/s
- 6 km (20000 ft), 10 kbit/s

Az ISO 11898 szabvány előírja a kábel hullámimpedanciával történő lezárását (csavart érpár esetén 120 ohm). A lezárás szerepe kettős: meggátolja a reflexiók kialakulását és biztosítja a helyes DC feszültségszinteket.

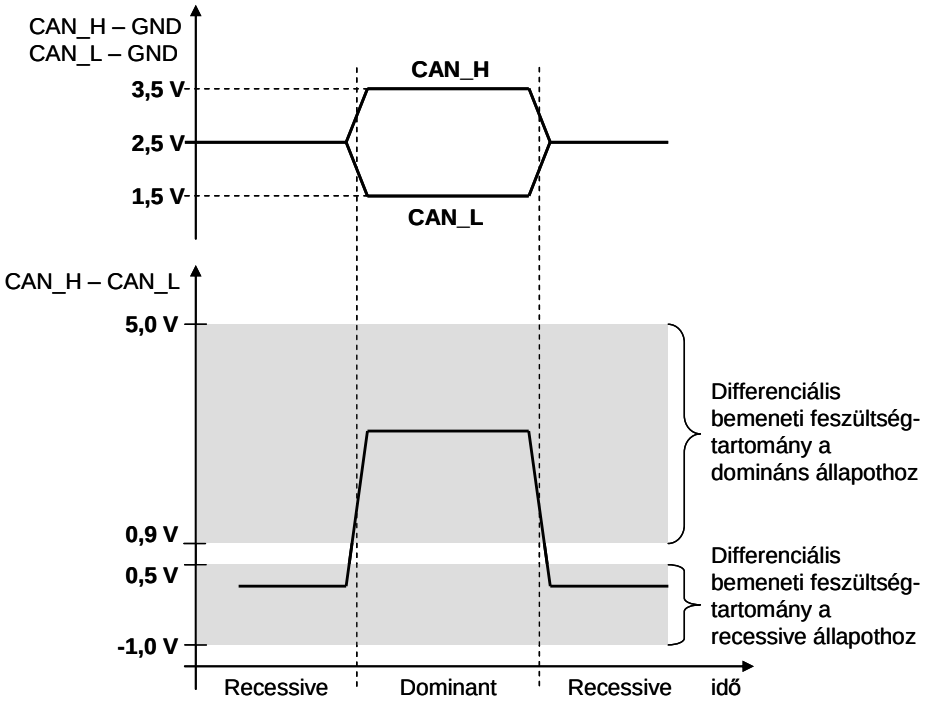
Érdekes módon a CAN szabvány nem adja meg az alkalmazandó csatlakozók típusát. A gyakorlatban a magasabb szintű protokollokban szokták rögzíteni a használandó

csatlakozókat (ez teljesen ellentmond az OSI hivatkozási modell koncepciójának). Legtöbbször 9-pólusú, DB-típusú csatlakozókat használnak.

Egy tipikus CAN meghajtó-vevő áramkör látható a 3.4 ábrán (a Philips 82C251 egyszerűsített rajza). Jól látható, hogy az adás és a vétel ugyanazon az érpáron történik (fél-duplex átvitel), továbbá a meghajtó a CAN_H vezetéket vagy elhúzza a tápfeszültség felé vagy elengedi, illetve a CAN_L vezetéket vagy elhúzza a föld felé vagy elengedi. A 3.5 ábrán ezek alapján már könnyen megérthetjük, hogy miért ilyenek a jelalakok és a jelszintek.



3.4 ábra. CAN transceiver egyszerűsített rajza

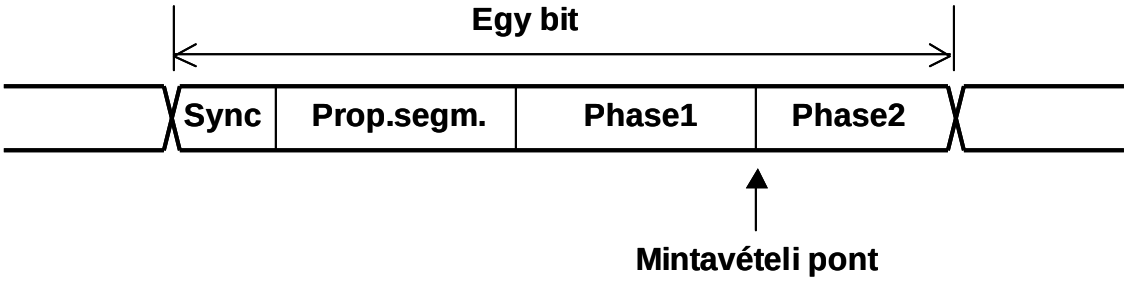


3.5 ábra. Feszültségszintek

Bitidőzítés

A CAN-buszon minden bit négy szegmensre van osztva, és minden egyes szegmens az ún. időkvantum (Time Quanta) egész számú többszöröse. A bitszegmensek a következők:

- Synchronization Segment
- Propagation Segment
- Phase Segment 1
- Phase Segment 2



3.6 ábra. A CAN bitidőzítése

A pontosan 1 kvantumnyi hosszúságú szinkronizációs szegmens a bitek szinkronizálását szolgálja. A buszon a jelváltásoknak ebben a szegmensben kell megtörténniük.

A jelterjedési (Propagation) szegmens a busz terjedési idejét kompenzálja (beleértve a kábel jelterjedési idejét és az elektronika, pl. optocsatolók késleltetéseit).

A mintavétel a fázisszegmensek (Phase Segment1 és 2) között történik. Ha szükséges, a mintavételi pont előre, hátra mozgatható, amit az egyik fázis rövidítésével és a másik nyújtásával érnek el (csökkentik vagy növelik a kvantumok számát az adott szegmensben).

Az utóbbi három szegmens hossza (beállítástól függően) egyenként 1 és 8 kvantum közötti, így egy bit legalább 4-25 időkvantum hosszúságú. Az időkvantumot a rendszerórából állítják elő leosztással.

Az órák szinkronizálása az 1-0 (recessive–domináns) átmeneteknél történik.

Hibajelzés

A CAN ötféle hibajelzési technikát használ:

- Bit Monitoring
- Bit Stuffing
- Frame Check
- Acknowledgement Check
- Cyclic Redundancy Check

Bit Monitoring – Adáskor az adó összehasonlítja a kiadott bitet a vett bittel. Ha az arbitrációs fázison kívül a két érték nem egyezik, Bit Error hiba keletkezik, amit az adó regisztrál. (Az arbitrációs fázisban a bitek eltérése üzemszerű az ütközések miatt.)

Bit Stuffing – Annak érdekében, hogy kellően sűrűn történjék jelváltás a buszon, a CAN bitbeszúrást alkalmaz az adó oldalon és bitkiejtést a vételi oldalon. Az adó minden ötödik

azonos értékű bit után beszúr egy ellentétes értékűt. A vevő figyeli a beérkező biteket, és a szabály megsértése esetén Stuff Error hibát jelez.

Frame Check – A CAN keret bizonyos bitjei rögzítettek, így az ettől való eltérést a vevők képesek észlelni. Ilyen típusú hiba esetén Form Error hiba keletkezik.

Acknowledgement Check – Egy sikeresen vett keretet a vevő nyugtáza, az ACK bitet nullába állítja (az adó 1, vagyis recessive értékkel adja ki). Ha egyetlen vevő sincs a buszon, akkor az ACK bit egyes értékű marad, ami az adóban Acknowledgement Error hibát generál. Az adó ily módon értesül arról, hogy vették-e a keretét. Azt természetesen nem tudhatja, hogy mindenki vette-e a keretét, csupán azt, hogy senki nem vette, vagy legalább egy állomás vette.

Cyclic Redundancy Check – Az üzenetek integritását 15-bites CRC-vel védik. CRC hiba esetén a vevő CRC Error hibát jelez.

Hibakezelés

A CAN fejlett hibakezeléssel rendelkezik. Minden egyes állomás tartalmaz egy Transmit Error Counter és egy Receive Error Counter számlálót, amelyeket inkrementálnak, ha hibát észlelnek, és dekrementálnak, ha hibátlanul tudnak adni. Az adáskor bekövetkező hibákat nyolcszoros súllyal számolják. Ha a hibák száma meghalad egy bizonyos értéket, az állomás a normális (Error Active) állapotból átmegy egy fokozott elővigyázatosságot biztosító (Error Passive) állapotba. Error Passive állapotban ha a hibák száma meghalad egy újabb küszöböt, az állomás nem adhat a buszra. A hibák megszűnése esetén, bizonyos szabályok betartásával, az állomás visszakerülhet normál üzemmódba.

4. LIN: Local Interconnect Network

A LIN első verzióját a Motorola dolgozta ki 1999-ben. Ennek sikeréből kiindulva létrehozták a LIN Konzorciumot 2000-ben olyan autóipari cégek közreműködésével, mint az Audi, BMW, Daimler Chrysler, Volkswagen, Volvo. Az első széles körben elterjedt verziót a LIN 1.2-t 2000 novemberében publikálták. A Konzorcium utolsó verziója a LIN 2.2A (2010.12.31.), amit felajánlottak az ISO-nak szabványosításra (DRAFT INTERNATIONAL STANDARD, ISO/DIS 17987-7; 2014-01-10; Road Vehicles – Local Interconnect Network (LIN)). A 2.2A és korábbi változatok ingyen letölthetők az Internetről, az ISO szabványokért viszont fizetni kell.

A LIN létrehozásánál egyértelműen arra törekedtek, hogy olcsóbb legyen, mint a CAN és más eddig használt kommunikációs technológiák, és albuszaiként legyen használva, mint ahogyan azt a Bevezető 1.3-as ábráján is láthatjuk.

A LIN 2.0 alapkoncepciója

A LIN alapkoncepciója, hogy a buszon egy master és több slave egység helyezkedik el, és mivel csak a master kezdeményezhet adatátvitelt, nem lépnek fel arbitrációs problémák. A LIN specifikáció által leírt soros kommunikáción alapuló protokoll az OSI modell két alsó rétegét tölti ki, tehát a fizikai és az adatkapcsolati réteget.

A LIN szabvány az alábbi részekből áll:

- Protokoll specifikáció (az adatkapcsolati réteg leírása)
- Physical Layer specifikáció (fizikai rétek: adatsebesség, órajel pontosság stb.)
- Diagnosztikai és konfigurációs specifikációk: Az adatkapcsolati réteg felett elhelyezhető funkciók leírása, például az egyes node-ok konfigurálásának folyamata.
- Configuration Language Specification: A LIN leíró file-ok áttekintése, amely segítségével az egész hálózati működés meghatározható. Ezeket használják a komplex LIN alapú rendszerek tervezésénél, és ilyeneket generálnak a magas szintű fejlesztő környezetek.
- Node Capability Language Specification: plug and play támogatás az egyes node-ok önleírásával.

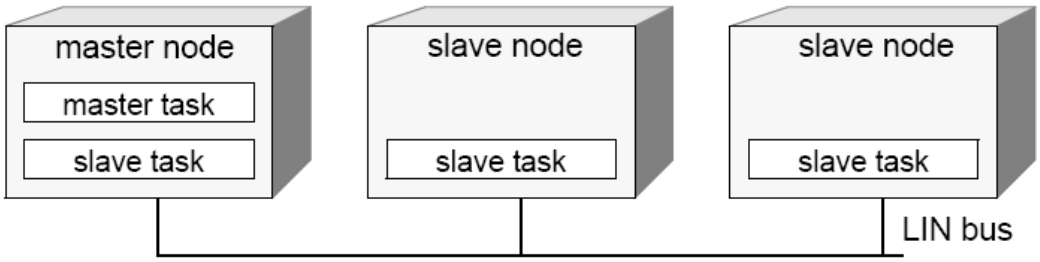
A LIN működési elve

A LIN aszinkron soros kommunikációra épül. Ennek fizikai implementációja egy egyvezetékes kommunikáció, amely half-duplex átvitelt tesz lehetővé. A LIN broadcast alapú, tehát a buszra egy egység által kiküldött információt az összes többi egység képes fogadni és feldolgozni. A LIN üzenetekbe speciális szinkronizáló mezőt illesztettek, amik lehetővé teszik a node-ok külső kvarcának elhagyását, ezzel is költségcsökkentést érve el az egyes node-oknál. Szintén a LIN egyszerűségére jellemző, hogy a kommunikáció megvalósítható a legtöbb mikrovezérlőben megtalálható normál UART perifériával.

Az adatkapcsolati réteg

Egy LIN hálózat egy masterből és több slave node-ból áll. A működés szempontjából tekintve pedig a rendszerben egy master és több slave taszk létezik. A master node hajtja

végre a master taszkot, illetve hajthat végre slave taszkokat is, amennyiben tartalmaz valamilyen perifériát. A LIN slave-ek pedig csak egy slave taszkot hajtanak végre.

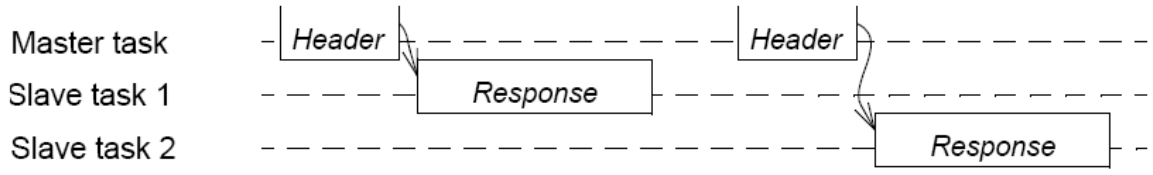


4.1 ábra. Egy LIN hálózat felépítése

A master taszk szabja meg, hogy a buszon milyen üzenet, mikor kerül elküldésre. Minden üzenetet a master indít el. A slave-ek csak akkor küldik el az adatukat, amikor erre a master felkéri őket. A masterben létezik egy ütemező tábla, hogy mikor, melyik slave-ket kell lekérdeznie. Normál ipari rendszerek esetében ennek a táblázatnak az összeállítására külön szoftvercsomagokat használnak fel, aminek az eredménye egy ún. LIN descriptor file, amiből a master node ütemező forráskódja automatikusan generálható.

Ez a működési mód determinisztikussá teszi az egész klaszter viselkedését, egy idővezérelt architektúrát létrehozva, ami a master ütemezőjétől függ. A buszon nincs arbitráció.

Az adatok kommunikációs keretekben kerülnek elküldésre. Egy általános kommunikációs keret áll egy headerből, amit a master taszk állít elő, és egy válaszból, amit a slave taszk ad ki (a masterben futó slave taszk válaszolhat az ugyanazon a node-on futó master taszk kérésére, ezáltal külső szemlélő számára a master node adhat ki komplett üzeneteket).



4.2 ábra. Egy LIN hálózati kommunikáció

Egy LIN üzenet fejléce áll egy **break**-ből egy **szinkronizációs szekvenciából** és egy **azonosítóból.** Erre a fejlécre az azonosító által megszólított slave task válaszol. A válasz tartalmazza az **adatmezőt** és egy **checksum mezőt.** Ha egy adat, amit a master által kiadott azonosító egyértelműen azonosít, fontos egy slave egység számára, akkor az a checksum ellenőrzése után felhasználhatja azt. Ez a következőket eredményezi:

- Úgy adhatunk hozzá új node-okat a rendszerhez, hogy a régiek viselkedését nem befolyásoljuk
- Egy üzenet célja és a tartalmazott információ jellege egyértelműen azonosítva van az ID által.
- Multicast rendszer, egy üzenetet több node használ fel.

- A LIN slave node-oknak fogalma sincs arról, hogy a háttérben milyen rendszerkonfiguráció helyezkedik el.
- A maximálisan elérhető azonosítók száma 64.

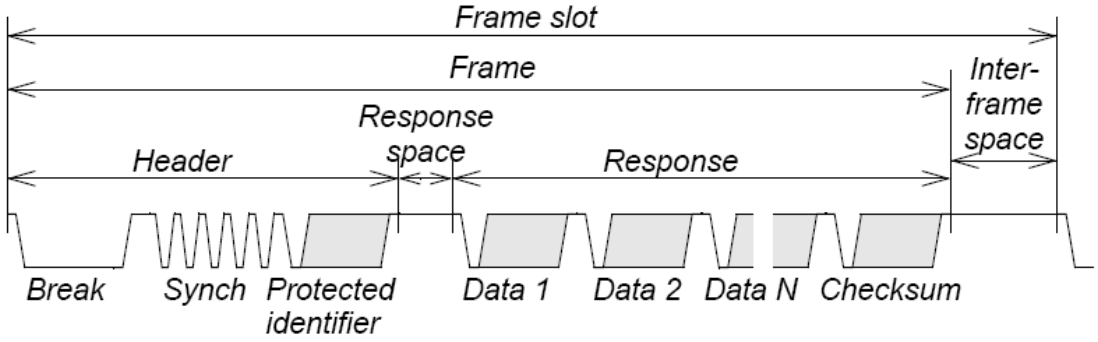
Adatcsomagok

A buszon két különböző jellegű adatot lehet átküldeni: signalokat és diagnosztikai információkat.

- **Signalok**: Fizikai paraméterek adatcsomagokba csoportosítva, például egy nyomógomb állapota. Egy signal az adott azonosítójú keretben mindig ugyanazokon az adatbájtokon helyezkedik el.
- **Diagnosztikai üzenetek**: A diagnosztikai üzeneteket speciális azonosítójú csomagok hordozzák.

Az adatcsomag felépítése

Az adatcsomag tartalmaz egy **break jelet**, majd azt követően egy 4-11 byte-os csomagot. Az adatbyte-ok a standard UART 8N1 formációban vannak ábrázolva, tehát egy adatbyte átvitelére 10 bit szolgál (1 startbit, 8 adatbit, 1 stopbit). Az adatbyte-oknál, az UART-oknál megszokott módon, az LSB-vel kezdődik az adatok átvitele.



4.3 ábra. Egy LIN keret felépítése

A fejléc részletesen

A fejlécet mindig a master küldi el. A fejléc az alábbi részekből áll:

- **Break signal**: minimum 13 bit hosszúságú domináns állapot.
- **Break delimiter**: minimum 1 bit hosszúságú recesszív állapot.
- **Synch, szinkronizációs mező**: egy 0x55 értékű byte. Ami a start és a stop bitet is beleértve 10 bitnyi 0 10101010 1 (start bit, hexa 55 LSB-MSB sorrendben, 1 stop bit) sorozatot állít elő a kommunikációs sebesség azonosítására és szinkronizációra. Néhány megjegyzés a szinkronizációval kapcsolatban: A LIN alapspecifikációja szerint a master node oszcillátor-toleranciája 0,5% alatti kell, hogy legyen. Azoknak a slave node-oknak pedig, amelyek nem rendelkeznek szinkronizációs mechanizmussal, 1,5%-os toleranciát kell nyújtaniuk. Azok a node-ok, amelyek rendelkeznek beépített szinkronizációval, elég, ha 14%-os toleranciát biztosítanak, de a szinkronizációs mező után 2%-ra be kell húzniuk magukat.

- **PID, Protected Identifier** (védett azonosító mező): a 8 bites PID egy 6 bites ID-ből és egy 2 bites paritásból áll. A lehetséges 64 címből a 60, 61 diagnosztikára, a 62, 63 későbbi bővítésre van fenntartva. Az üzenethossz (1-8 bájt) nincs explicit módon megadva, ezt a fejlesztők a programozáskor előre rögzítik. (Az 1.1 változatban az első 6 bit értelmezése még más volt: 4 bit üzenetazonosító (ID), 2 bit üzenethossz (0 = 2 bájt, 1 = 2 bájt, 2=4 bájt, 3=8 bájt).)

Az adatmező

Az adatmező az 1.1 változat szerint 2, 4 vagy 8 byte-ot tartalmazhat, a magasabb változatokban 1-8 bájtot.

A checksum byte

A keret ellenőrző összege.

Nyugtázás

A LIN nem tartalmaz nyugtázást. A master a slave válaszokból deríti ki, hogy nyugtázták-e a csomagját.

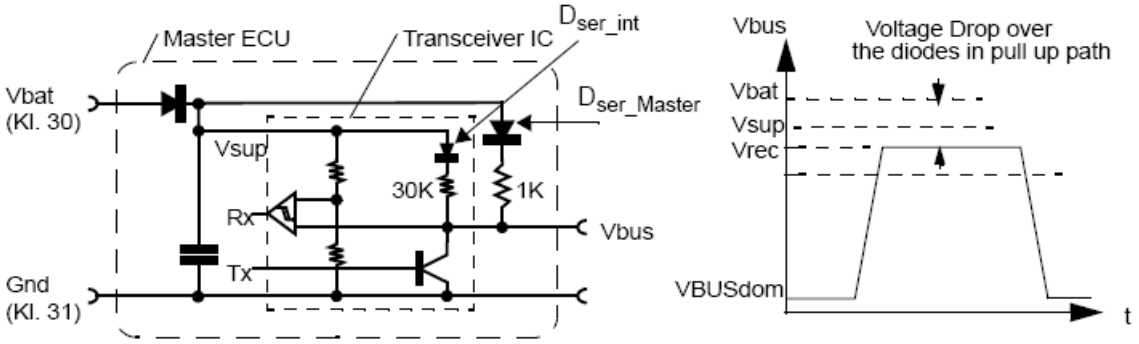
Hibadetektálás

A küldőnek mindig monitoroznia kell, hogy a buszon valóban az általa kiadott jelszintek jelennek-e meg. A keretekben is van hibadetektálási rész: a 2 paritásbit az ID végén és a CRC mező az üzent végén.

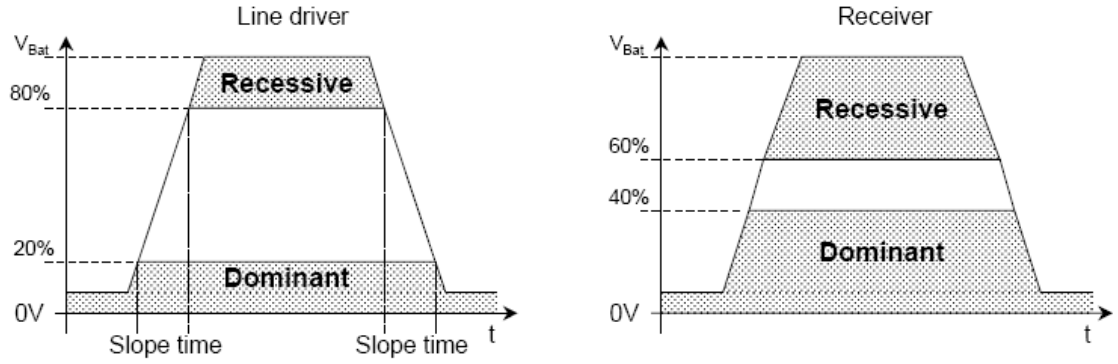
Hibajelzés

Az aktív hibajelzés nem valósítható meg ebben az architektúrában, ugyanakkor az egyes slave-ektől lekérdezhetők az általuk észlelt hibák a diagnosztikai üzenetek felhasználásával.

A fizikai réteg



4.4 ábra. Egy LIN meghajtó egyszerűsített felépítése



4.5 ábra. LIN jelszintek

A LIN fizikai rétege az ISO 9141 továbbfejlesztése (K-Line). A busz meghajtása open-collectoros jellegű (4.4 ábra), és az autóakkumulátor szintjéhez (V_{bat}) igazodik (4.5 ábra).

- **Kommunikációs sebesség**: A maximális adatsebesség 20 kbit/s, de általában a standard UART bitsebességeket választják (300, 600, 1200, 2400, 4800, 9600, 19200). Szintén elterjedt sebesség a K-Line által használt 10,4 kbit/sec.
- **Bitkódolás:** NRZ (Non Return to Zero).
- **A busz értékek:** domináns érték 0, recessive 1.
- **Javasolt node szám:** Az ajánlott maximális node szám 16.
- **Vezetékhosszúság:** max. 40 m
- **Lezáró ellenállás:** 1 kOhm master, 20-47 kOhm slave.
- **Buszkapacitás:** A master buszkapacitásának szignifikánsan nagyobbnak kell lennie, mint a slave-ekének, azért, hogy új slave-ek rácsatlakozása miatti változások ne érződjenek a hálózaton.

LIN interface

A D1 dióda azért fontos, hogy megakadályozzuk a node buszon keresztüli táplálását abban az esetben, ha elveszítené a földet. A D2 a tápellátás fordított bekötése ellen véd.

LIN fizikai réteggel kapcsolatos kompromisszumok

- Minden node saját oszcillátorral rendelkezik, de a hőmérséklet-különbség nagyon nagy lehet a node-ok között.
- A buszra kapcsolt kapacitás rontja a sebességet, de szükséges a rádiófrekvenciás zavarok kiszűrésére.
- Az egyes node-ok különböző kapacitív és rezisztív terhelései teljesen más felfutási időket eredményezhetnek node-onként.
- EMC problémák: az aszimmetrikus kommunikációból adódóan egy 15-20 kbit/s-os LIN busz hozzávetőlegesen akkora EMC terhelést okoz, mint egy 500 kbit/s-os CAN busz.

5. CAN mérési feladatok

Ebben a mérésben megismerkedünk a CAN alapjaival, egy CAN protokollanalizátor használatával, és bepillantást nyerhetünk az autókban alkalmazott elektronikus egységek világába. Oszcilloszkóppal megvizsgáljuk a CAN fizikai rétegét (jelszintek, jelalakok), majd oszcilloszkóppal és protokollanalizátorral elemezzük az adatkapcsolati réteg keretformátumát. Az alkalmazási réteget – ebben az esetben egy autós rendszer belső kommunikációját – szintén protokollanalizátorral fogjuk elemezni. A mérés végén Bosch gyártmányú valódi autós perifériákból, vezérlő egységekből, játékperifériákból és játékprogramból összeállítunk egy drive-by-wire autómodellt, amelynek működését a CAN-busz monitorozásával fogjuk nyomon követni, naplózni és utólag feldolgozni.

Mérési feladatok

1. A CAN busz fizikai rétegének vizsgálata

a. Mérje meg a CAN-buszon a feszültségszinteket (a CAN_H és CAN_L vezetékeket egymáshoz képest és a GND-hez képest)! Vizsgálja meg a jelalakokat is!

b. Határozza meg oszcilloszkóp segítségével a CAN-busz sebességét!

c. Hogyan változnak a jelalakok különböző hosszúságú buszkábelek és lezárások esetén?

d. Mi történik, ha nincs lezárva a busz?

2. Egy CAN üzenet visszafejtése digitális oszcilloszkóp segítségével

Fejtse vissza oszcilloszkóppal a mérésvezető által a CAN-buszra adott keret mérésvezető által meghatározott részét (pl. az ID-mezőt)!

3. Ismerkedés a CANalyzerrel

a. Hajtsa végre A CANalyzer használata segédletben leírt alapfeladatokat!

b. Fejtse vissza az analizátorral a mérésvezető által megadott CAN változókat (signalokat), valamint a kormány és a gáz- és fékpedál üzeneteket!

4. CAN periféria programozása

Készítsen egy olyan programot a mitmót alapú fejlesztőkártyára, amely segítségével elvezethető a TORCS szimulált autója a mitmót nyomógombjai segítségével.

5. Komplex mérési feladat a CANalyzer használatával

a. A mérőcsoport valamennyi tagja menjen egy-egy kört a drive-by-wire autóval a szimulált versenypályán, és naplózza az autó megfelelő paramétereit!

b. A naplózott adatokból mindenki számítsa ki a mérésvezető által megadott paramétereket (pl. maximális sebesség, átlagsebesség, pillanatnyi gyorsulás/lassulás, a sebességváltások számát)!

6. LIN mérési feladatok

Ebben a mérésben megismerkedünk a LIN alapjaival. Oszcilloszkóppal megvizsgáljuk a LIN fizikai rétegét (jelszintek, jelalakok), majd oszcilloszkóppal és protokollanalizátorral elemezzük az adatkapcsolati réteg keretformátumát. Elemezzük az alkalmazási réteget, ebben az esetben egy minta body alhálózat belső kommunikációját. Megismerkedünk egy LIN slave programozásával, majd létrehozunk egy olyan komplex LIN rendszert, amely kapcsolatot tart egy gyorsabb CAN gerinchálózattal és annak az információit is felhasználja a működése során.

Mérési feladatok

1. A LIN busz fizikai rétegének vizsgálata

a. Mérje meg a LIN-buszon a feszültségszinteket!
b. Határozza meg oszcilloszkóp segítségével a LIN-busz sebességét, és számítsa ki annak eltérését a névlegestől!

2. Egy LIN üzenet visszafejtése digitális oszcilloszkóp segítségével

Fejtse vissza oszcilloszkóppal a minta LIN rendszer egy üzenetét!

3. Egy LIN slave tanulmányozása

Ellenőrizze a mintarendszerben használt LIN slave működését, perifériakezelését oszcilloszkóp segítségével.

4. Ismerkedés a CANalyzer LIN opciójával

Hajtsa végre *A CANalyzer használata* segédletben leírt LIN alapfeladatot!

5. LIN slave programozása

Egészítse ki az egyik LIN slave programját a mérésvezető által megadott módon!

6. Komplex LIN rendszer létrehozása. Válasszon feladatot az alábbiak közül!

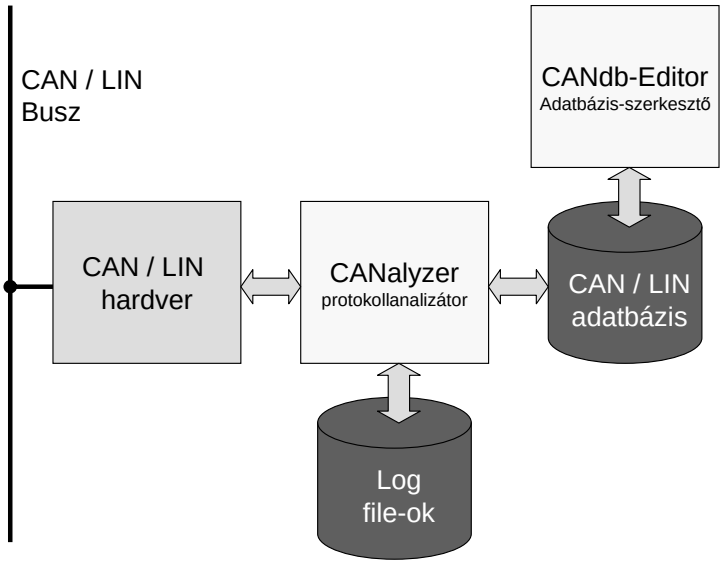
a. Hátsó lámpablokk megvalósítása: A féklámpa kigyújtása, ha a CAN buszon a fékpedál állását tartalmazó jel értéke nem nulla. Az index bekapcsolása a lokális LIN buszon lévő kapcsoló hatására.
b. Központi zár megvalósítása: A központi zár motorjának mozgatása a lokális LIN buszon elhelyezkedő kapcsoló hatására. A központi zár automatikus bezárása, ha az autó sebessége 20 km/óra fölé emelkedik.

7. Függelékek

7.1 A CANalyzer használata CAN és LIN hálózatok megfigyelésére

A CANalyzer egy olyan általános protokollanalizátor, amelyet kimondottan az autóiparban használt buszokhoz fejlesztettek ki. A CANalyzer alapvetően a CAN protokollt ismeri, de rendelhetők hozzá kiegészítő hardvermodulok és szoftvercsomagok, amelyek a LIN és akár FlexRay protokollok forgalmának analizálását is lehetővé teszik.

A CANalyzer segítségével nyomon követhetjük és akár későbbi analízis céljára eltárolhatjuk a kommunikációs busz forgalmát. A CANalyzer alkalmas arra, hogy grafikusan megjelenítsük egyes változók értékét, szűrő feltételeket állítsunk be a venni kívánt üzenetekre, valamint hogy üzeneteket generáljunk a segítségével (akár teljes log-okat is visszajátszhatunk). A protokollanalizátor további funkciója, hogy a buszon folyó kommunikációról részletes statisztikát készítsen, valamint jelezze az esetleges kommunikációs hibákat.

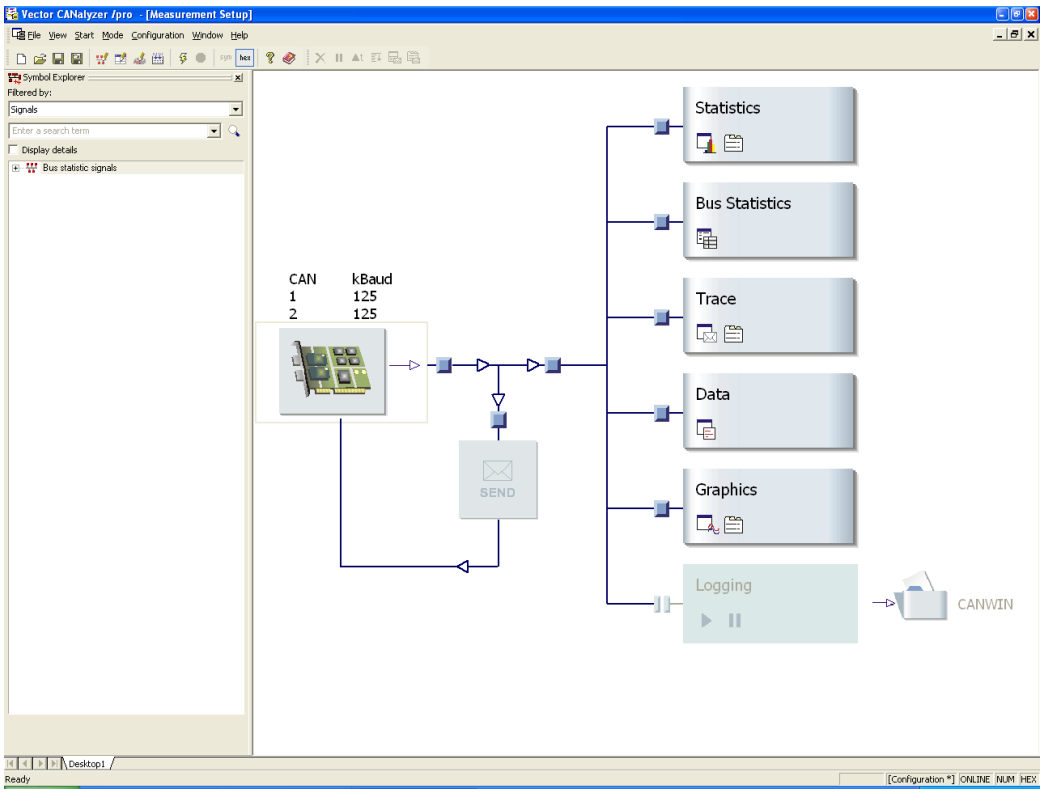


7.1. ábra. A CANalyser felépítése

A mérésben használt protokollanalizátor főbb egységeit a 7.1 ábrán láthatjuk. A CAN/LIN hardver – a mi esetünkben egy PCI-os vagy USB-s kártya két interfésszel – látja el a buszcsatlakozás funkcióját. Ez gyakorlatilag a fizikai réteget valósítja meg, továbbá néhány magasabb szintű funkciót tartalmaz, mint például hibavédelem és CAN esetében az arbitráció. A CANalyzer a kártya segítségével tud csomagokat küldeni és fogadni. A CANalyzer szoftverblokk tartalmazza az analizátor funkciót; ennek a programnak a segítségével tudjuk az előzőekben bemutatott mérési, beavatkozási funkciókat megvalósítani. A CANalyzer a háttértárra tárolja el a mérésekről készített napló (log) file-okat, illetve ami ennél lényegesen érdekesebb és fontosabb, hogy szintén külső file-okból tölti le az ún. CAN vagy LIN adatbázist. A CAN vagy LIN adatbázis – amelyet egy külön program, a CANdb-Editor segítségével hozhatunk létre – tartalmazza, hogy az egyes CAN vagy LIN azonosítókhoz milyen üzenet tartozik, illetve, hogy ezek az üzentek milyen ún. signal-okat tartalmaznak. Ez a leíró file azért nagyon fontos, mert a CAN és a LIN sem ad direkt módon utasítást az egyes

azonosítók felhasználási módjára, ezért ezek alkalmazásfüggők. Ahhoz, hogy a protokollanalizátor felismerje a kommunikációban használt egyes változókat, meg kell adnunk, hogy az a változó melyik üzenetben és azon belül hol helyezkedik el (a későbbiekben erre láthatunk egy példát).

A CANalyzer felépítése után ismerjük meg a használatát is! Indítsuk el a programot! Az elindulás után a program betölti a legutoljára használt mérési konfigurációt, de ezt most hagyjuk figyelmen kívül, és hozzunk létre egy teljesen új konfigurációt (*File/New configuration…*)! Az új konfiguráció létrehozásához használjuk a *default template* mintát (CAN esetében a CAN-hez tartozót, LIN esetében a LIN-hez tartozót)! Ennek hatására a program létrehozza az új üres mérési konfigurációt, és feldob egy csomó ablakot. Ahhoz, hogy a mérési konfigurációt helyesen beállítsuk, a *Measurement Setup* ablakra (7.2 ábra) lesz szükségünk, a többi frissen megjelent ablakot zárjuk be (*Data, Bus Statistics, Write, Graphics, Trace, Statistics*)!

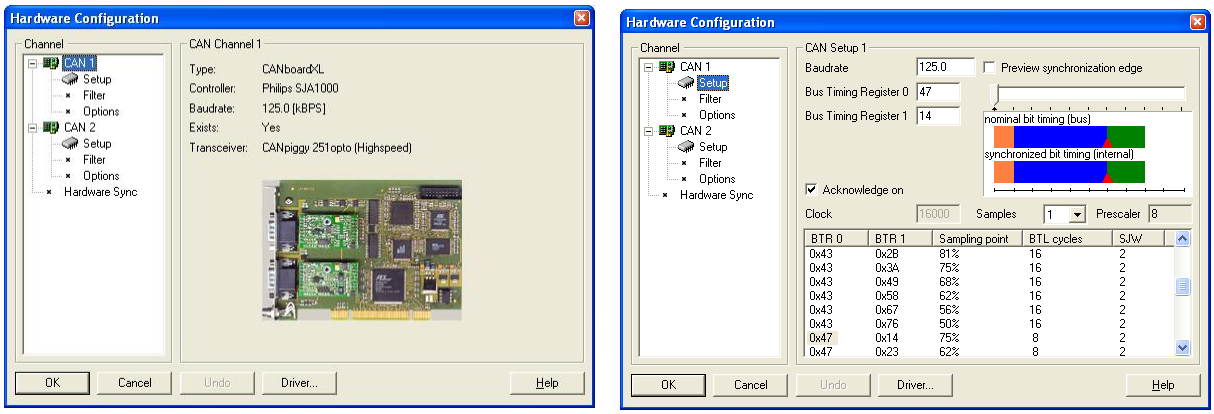


7.2 ábra. A Measurement Setup ablak

A *Measurement Setup* ablak gyakorlatilag a CANalyzer fő konfigurációs menüje. Itt állíthatjuk be a buszcsatlakozás fizikai paramétereit, a vételnél használt szűrőket, a buszra elküldeni kívánt adatokat, illetve hogy a vett adatokat milyen feldolgozásnak vetjük alá. A 4.2 ábrán a mérési összeállítás adatfolyamát láthatjuk (az ábrán CAN csatornákat láthatunk, de LIN esetében is nagyon hasonló képet kapunk). Minden funkcionális blokk egy összeköttetést jelentő tömör négyzettel (gyakorlatilag a *Logging* kivételével az összes blokk) vagy egy szakadást jelentő kondenzátorszerű ikonnal csatlakozik a mérési folyamathoz (lásd a *Logging* blokkot). Bármely blokkot dinamikusan hozzácsatlakoztathatjuk vagy leválaszthatjuk a mérési elrendezésről úgy, hogy az összeköttetést vagy szakadást szimbolizáló ikonra kattintunk. Az éppen inaktív blokkok mindig halványan kerülnek megjelenítésre. A csomópontokra jobb gombbal kattintva olyan új modulok is beiktathatók a mérési elrendezésbe, mint például szűrő- és triggerblokkok.

Az új mérési elrendezés konfigurálását mindig a bal oldalon található fizikai illesztők beállításával kezdjük.

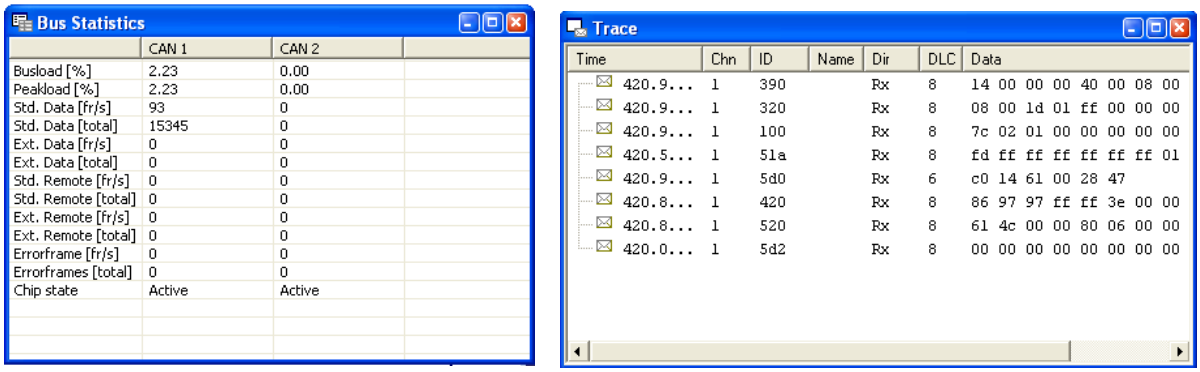
CAN esetében konfiguráljuk a fizikai interfészt 500 kBaudos sebességre, mert a Laborautó és a váltóvezérlő is ezt a sebességet használja (7.3 ábra)!



7.3 ábra. A CAN-illesztő konfigurálása

LIN esetében ehhez nagyon hasonló módon állítsuk be a kommunikáció sebességét 19200 Baudra, illetve a LIN szabványt 1.3-ra (mással is működne).

A fizikai csatlakozás helyes konfigurálása után már el is kezdhetünk mérni a CANalyzer-rel. Indítsunk is el egy próbamérést: kattintsunk a *Bus Statistics* modulra, majd indítsuk el a mintavételezést (*Start/Start* vagy *F9* vagy a menük alatt található villámot mintázó ikon segítségével)! A *Bus Statistics* ablakban (7.4. ábra balra) meg fog jelenni, hogy az egyes interfészeken milyen fogalom zajlott a mintavételezés elindítása után. Amennyiben arra is kíváncsiak vagyunk, hogy milyen adatok kerültek ki a buszra, a *Trace* ablakot (7.4. ábra jobbra) kell használnunk.



7.4 ábra. Mérési adatok megjelenítése a Bus Statistics *és a* Trace *ablak segítségével*

A *Trace* ablak információjának megjelenését kétféleképpen konfigurálhatjuk. Először is a *Measurement Setup* ablakban láthatjuk, hogy a *Trace* blokkhoz két ikon is tartozik. Általános szabályként azt mondhatjuk, hogy az első ikon az adatmegjelenítő, a második pedig a konfigurációs funkciót indítja el (próbáljuk ki). További konfigurációs lehetőséget nyújtanak a menüsor alatt található ikonok.

Mint láthatjuk, a *Trace* ablak csupa nyers információt szolgáltat a számunkra, és emiatt azok értelmezése nagyon nehézkes. Ahhoz, hogy a felhasználó számára könnyebben értelmezhető adatok jelenjenek meg, illetve hogy a grafikus megjelenítési opciókat használni tudjuk, szükségünk van a már említett CAN vagy LIN leíró fájlra. Mintaként itt bemutatjuk,

hogy hogyan kell elkészíteni CAN esetében egy ilyen leíró fájlt (LIN esetében gyakorlatilag ugyanezt a műveletet kell végrehajtanunk, sőt még az egyes konfigurációs ablakok is tökéletesen azonosak).

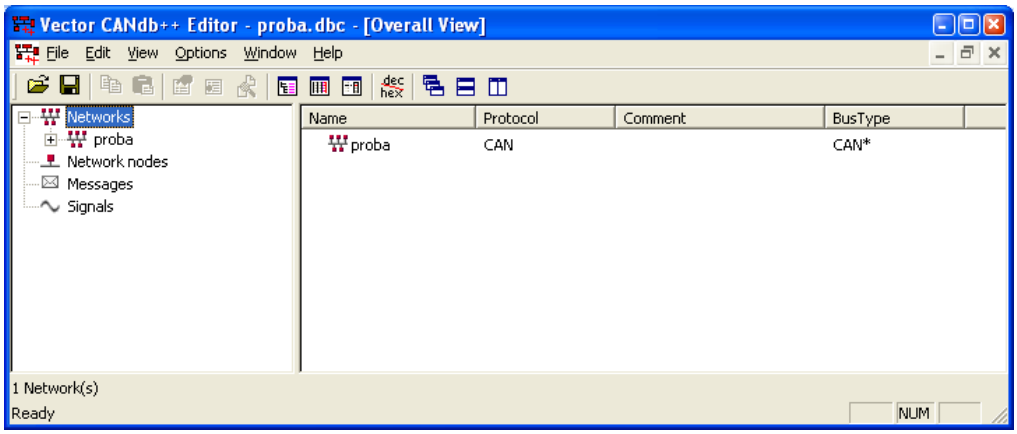
CAN ID: 0x320 Kiegészítő sebesség
Ezt az üzenetet az autós műszerfal küldi ciklikusan.

Hossz: 8 byte
Signal lista:

Speed

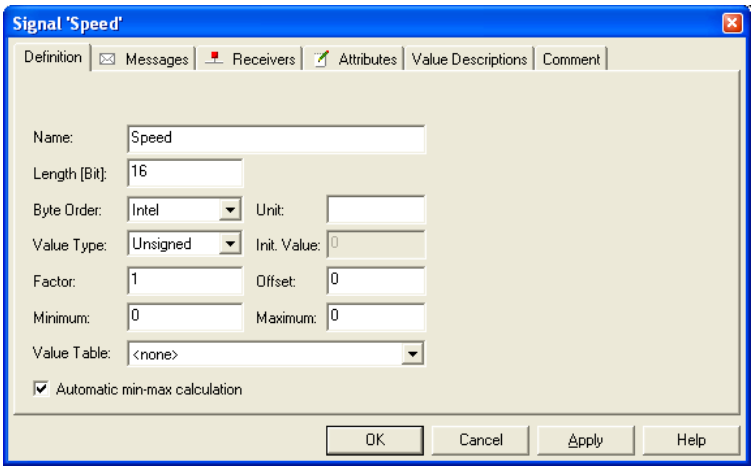
Start bit:	40. bit
Length:	16 bit
Byte order:	Little endian (Intel)
Value type:	Unsigned

Első lépésként indítsuk el a *CANdb-Editor*-t (*File/Open CANdb Editor*), majd hozzunk létre egy új adatbázist (*File/Create Database* és válasszuk a *CANTemplate*-et)!



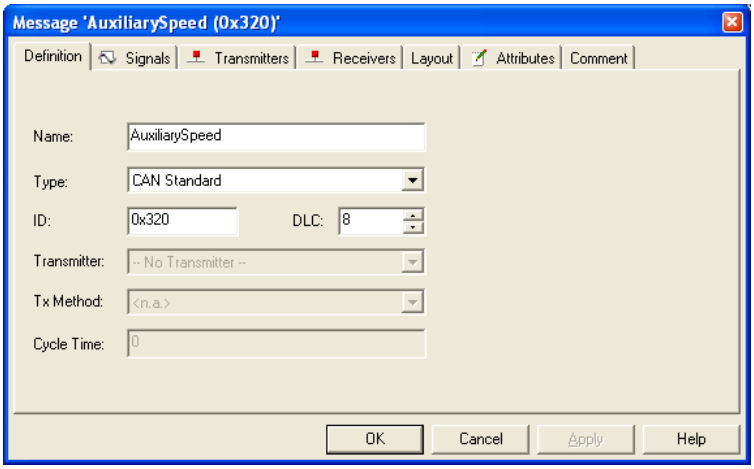
7.5 ábra. A CANdb-Editor

A 7.5. ábrán látható ablak fog megjelenni. Először is hozzunk létre egy új *Signal*-t (változót) úgy, hogy a *Signal* fülön jobb gombot nyomunk, majd kiválasztjuk a *New* menüpontot. A megjelenő ablakban értelemszerűen töltsük ki az ismert mezőket (*Length:* 16bit, *Byte Order*: Intel, *Value type*: Unsigned; a *Factor*-t és az *Offset*-et egyelőre nem ismerjük).



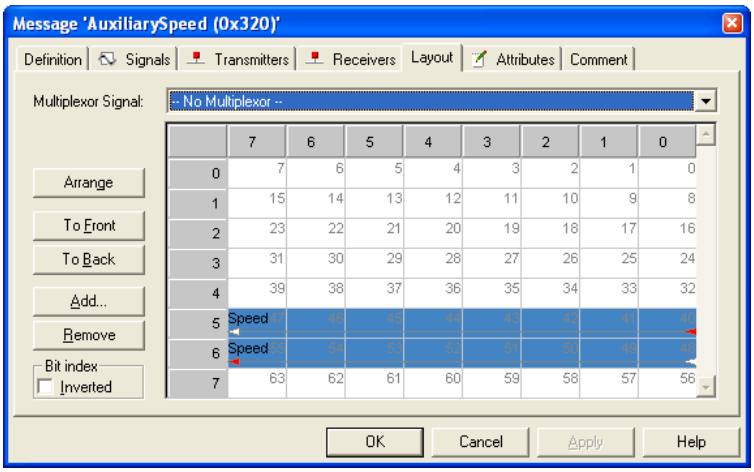
7.6 ábra. Új Signal létrehozása

Az újonnan létrehozott *Signal* megjelenik a listában, és ezek után hozzáadhatjuk az egyes üzenetekhez. Ezt tegyük is meg, de ehhez először is egy új üzenetet (*Message*) kell létrehoznunk a *Signal*-hoz hasonló módon (a 7.6. ábra nyújt segítséget a létrehozáshoz).



7.7 ábra. Új Message létrehozása

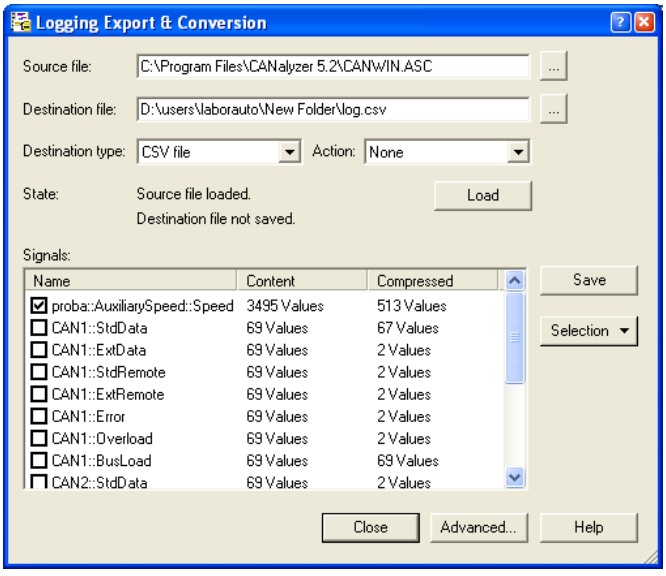
Miután az új üzenet általános paramétereit megadtuk, adjuk hozzá a *Speed Signalt*, ezt a *Signal* fül segítségével tehetjük meg (*Add* ikon, és válasszuk ki az előzőekben létrehozott S*ignal*-t). Az utolsó teendőnk pedig az, hogy megadjuk, hogy ez a *Signal* hol helyezkedik el az üzeneten belül, erre a *Layout* fület tudjuk használni. Itt egyszerűen megfoghatjuk és a helyére húzhatjuk a változónkat.



7.8 ábra. A Layout beállítása

Mentsük el az adatbázisunkat, és adjuk hozzá a mérési konfigurációhoz! Ha éppen fut egy mérés, akkor állítsuk le, majd a *File/Associate Database* menüpont segítségével adjuk hozzá a konfigurációhoz a CAN leíró file-t! Tapasztalni fogjuk, hogy egy új mérés indításakor már a 0x320-as üzenetet felismeri a rendszer. Egy új mérés segítségével állítsuk be a *Speed* változó skálaértékét helyesen, majd ábrázoljuk a változó értékét a *Graphics* modul segítségével! Ehhez mindössze annyit kell tennünk, hogy a *Graphics* ablakhoz *jobb gomb/Add Signals* segítségével hozzáadjuk a megjeleníteni kívánt változót.

A mérés során fontos lehet, hogy tudjunk napló- (log) fájlokat készíteni. A log fájl elkészítéséhez csatoljuk hozzá a *Logging* blokkot a méréshez!. A *Logging* blokknál be tudjuk állítani, hogy mikor induljon a mérés, a mérés végén pedig a *Logging* blokktól jobbra lévő rész segítségével tudjuk kiválasztani, hogy mely mért paramétereket exportáljunk ki. A mérés után *jobb gomb/Export*, erre megjelenik a konfigurációs ablak (7.9. ábra).



7.9 ábra. A Logging beállítása

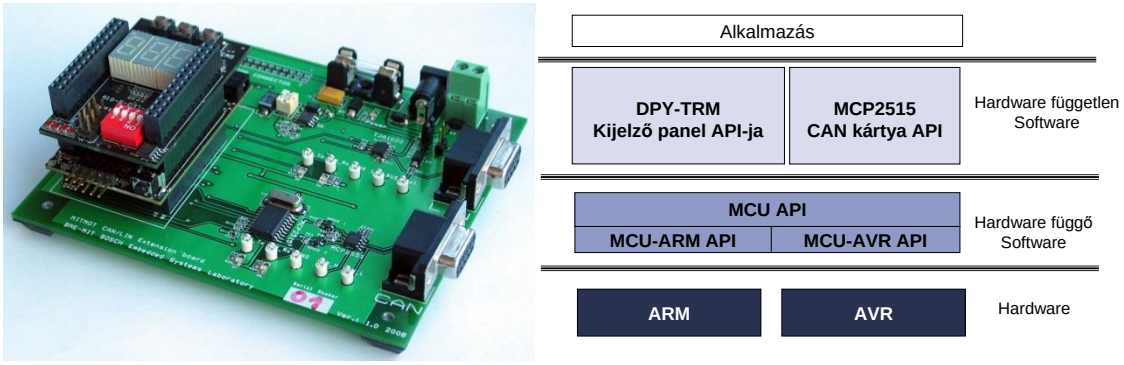
A *Source file* a CANalyzer által automatikusan létrehozott CANWIN.ASC lesz, a *Destination file*-t adjuk meg, majd a *Load* gomb segítségével töltsük be a log fájlban tárolt változókat, és jelöljük ki az általunk exportálni kívántakat! Az *Advanced..* részben beállíthatjuk az oszlopok szeparációjának módját, illetve a tizedespont/tizedesvessző

megjelenítési formáját. A megfelelő beállítások megadása után a *Save* gombbal menthetjük el az adatokat.

Az ebben az útmutatóban leírtak a CANalyzer használatának csak egy csekély, célirányos részét tartalmazzák. További információkat a weben találhatunk a programról.

7.2 Mintakód a CAN interfész programozásához

A laborban a *Mitmót CAN/LIN Extension board*-ot (7.10 ábra) fogjuk használni a CAN hálózatra való csatlakozáshoz. Ezen a kártyán a szokványos mitmót API-k ugyanúgy használhatók, mint a normál *mitmót* konfigurációkon. A különbség csak annyi, hogy a standard funkciók ki lettek egészítve egy a CAN csatlakozást lehetővé tevő a Microchip által gyártott MCP2515-ös CAN controller chippel. Ennek a chipnek a kezeléséhez ugyanúgy nyújtunk API támogatást, mint a többi kártyához.



7.10 ábra. Mitmót CAN/LIN Extension board és a hozzá tartozó API támogatás

Ez az API támogatás a gyakorlatban annyit jelent, hogy például a kijelző panel nyomógombjainak, kapcsolóinak és kijelzőjének kezelésére előre megírt eljárásaink vannak, amiket csak az adott paraméterekkel meg kell hívnunk. Ugyanilyen függvénytámogatásunk van a CAN kommunikációhoz is. Nézzük meg, hogy milyen függvényekre lehet szükségünk a labor folyamán!

Nyomógomb- és kapcsolókezelő függvények

A nyomógombokat és kapcsolókat kezelő függvények definícióját a *dpy_trm_s01.h* file-ban találjuk meg, ami a kijelző API header file-a. Felhasználható függvények:

DPY_TRM_S01__SWITCH_n_GET_STATE

Szintakszis

```
DPY_TRM_S01__SWITCH_n_GET_STATE();
```

`n` egy kapcsoló sorszáma [1..4]. A számozás a modulról leolvasható.

Meghívásának előfeltétele

`dpy_trm_s01__Init()` meghívása.

Bemenő paraméterek

Nincsenek.

Válasz paraméterek

Az `n.` kapcsoló állapotát adja vissza (ON állásban 1, különben 0).

DPY_TRM_S01__BUTTON_n_GET_STATE

Szintakszis:

```
DPY_TRM_S01__BUTTON_n_GET_STATE();
```

`n` egy nyomógomb sorszáma [1..3]. A számozás a modulról leolvasható.

Meghívásának előfeltétele:

`dpy_trm_s01__Init()` meghívása.

Bemenő paraméterek:

Nincsenek.

Válasz paraméterek:

Az `n.` nyomógomb állapotát adja vissza (ha a gomb nincs lenyomva 1, különben 0).

dpy_trm_s01__7seq_write_number

Szintakszis:

```
unsigned char dpy_trm_s01__7seq_write_number(
                float number, unsigned char decimal_fraction);
```

Meghívásának előfeltétele:

`dpy_trm_s01__Init()` meghívása.

Bemenő paraméterek:

`float number,` a megjelenítendő lebegőpontos szám

`unsigned char decimal_fraction` a tizedes jegyek száma

Válasz paraméterek:

Ha az adott feltételekkel a szám nem jeleníthető meg (lásd a függvény működésénél), `DPY_TRM_S01_7SEG__ERROR`, különben `DPY_TRM_S01_7SEG__NOERROR` a visszatérési érték.

A függvény működése:

Ellenőrzi, hogy a bemenetként kapott szám a megjeleníthető tartományba esik-e:
-99…999 `decimal_fraction`=0 mellett,
-9.9…99.9 `decimal_fraction`=1 mellett és
0…9.99 `decimal_fraction`=2 esetén.
Ha a bemenet nem ebbe a tartományba esik, a függvény `DPY_TRM_S01_7SEG__ERROR` hibajelzéssel visszatér, különben pedig `dpy_trm_s01__7seq_write_3digit` megfelelő hívásával kiírja a számot.

A kijelző panel többi függvényének részletes leírását a megfelelő mitmót API dokumentáció tartalmazza.

CAN kommunikációs függvények

Ennek az API-nak a segítségével lehetőségünk nyílik CAN üzenetek küldésére és fogadására. A függvények definícióit az mcp2515.h fájl tartalmazza. Fontosabb függvényei következők:

can_send_standard_message

Szintakszis

```
void can_send_standard_message(CAN_message *p_message);
```

A CAN_message struktúrában található üzenetet elküldi a CAN buszon keresztül.

Meghívásának előfeltétele

`mcp2515_init()` meghívása.

Bemenő paraméterek

`CAN_message *p_message`

Az alábbi szerkezetű struktúra, amely tartalmazza a CAN üzenet egyes részeit.

```
typedef struct
{
    unsigned short int        id;
    unsigned char             rtr;
    unsigned char             length;
    unsigned char             data[8];
} CAN_message;
```

Válasz paraméterek

Nincsenek.

can_receive_message

Szintakszis

```
void can_receive_message (CAN_message *p_message);
```

Addig várakozik, amíg a CAN buszon nem érkezik egy üzenet. Ekkor ezt az üzenetet a CAN_message struktúrába letárolja.

Meghívásának előfeltétele

`mcp2515_init()` meghívása.

Bemenő paraméterek

`CAN_message *p_message`

Az alábbi szerkezetű struktúra, amely tartalmazza a fogadott CAN üzenet egyes részeit.

```
typedef struct
{
    unsigned short int        id;
    unsigned char             rtr;
    unsigned char             length;
    unsigned char             data[8];
} CAN_message;
```

Válasz paraméterek

Nincsenek.

Mivel a CAN buszon számos, különböző ID-vel rendelkező üzenet található, ezért az API tartalmaz olyan függvényeket is, amelyekkel szűrési feltételeket lehet adni a fogadandó csomagokkal kapcsolatban, de ezekkel egyelőre nem kell foglalkoznunk.

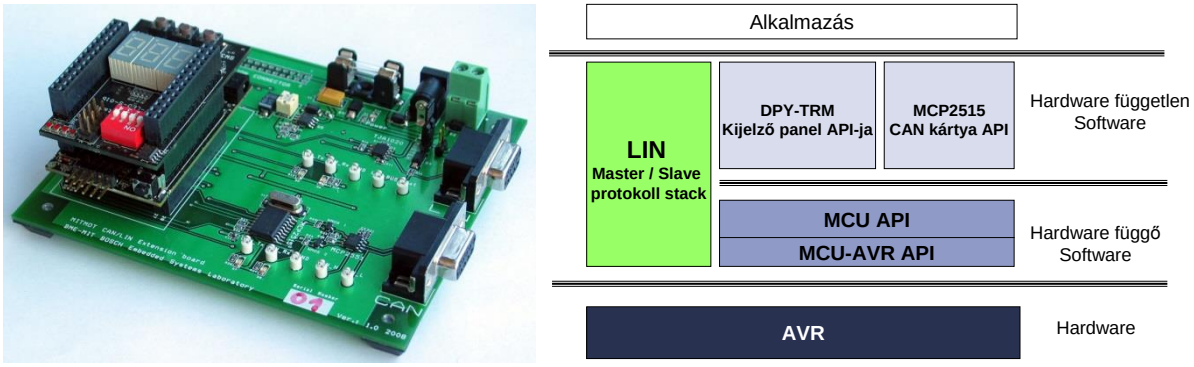
Mintakód CAN kommunikációra

Az alábbi egyszerű mintakód elküld egy 0x123 ID-jű üzenetet a CAN buszon, majd várakozik egy üzenetre, és annak a harmadik adatbyte-ját kijelzi a hétszegmenses kijelzőre.

```
#include "platform.h"
#include "mcp2515.h"
#include "dpy_trm_s01.h"

void main()
{
CAN_message message,rx_message;

      dpy_trm_s01__Init();    // A kijelző panel iniciálása
      mcp2515_init();         // A CAN kommunikáció iniciálása

      message.id = 0x123;
      message.rtr = 0;
      message.length = 2;
      message.data[0] = 0;
      message.data[1] = 0;    // Egy CAN üzenet összeállítása

      while(1)
            {
                  message.data[0]++;
                  can_send_standard_message(&message); // CAN üzenetküldés
                  _delay_ms(50);
                  can_receive_message(&rx_message);   // CAN üzenetfogadás
                  dpy_trm_s01__7seq_write_number(rx_message.data[2], 0);
            }
}
```

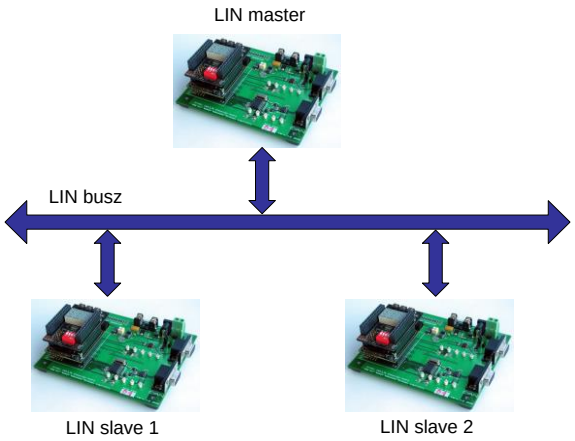
7.3 Mintakód a LIN interfész programozásához

A laborban a *Mitmót CAN/LIN Extension board*-ot (7.11 ábra) fogjuk használni a LIN hálózatra való csatlakozáshoz. Ezen a kártyán a szokványos mitmót API-k ugyanúgy használhatók, mint a normál *mitmót* konfigurációkon. A különbség csak annyi, hogy a standard funkciók ki lettek egészítve egy a LIN csatlakozást lehetővé tevő LIN protokollcsomaggal. A LIN protokollcsomag az Atmel cég AVR322 application note-ján alapul.



7.11 ábra. Mitmót CAN/LIN Extension board és a hozzá tartozó API támogatás, valamint a LIN protocol stack

A LIN protokollkészlet működése röviden összefoglalva a következő. A protokollkészlet működése gyakorlatilag független az alkalmazói programtól, mert teljesen eseményvezérelt. A LIN stack két külső eseményt ismer: az UART periférián érkező Rx (tehát vételi) és az AVR processzor egyik timere által kiváltott megszakítást. E megszakítások hatására a protokollkészlet automatikusan ellátja a különböző üzenetfogadási és küldési feladatokat. A felhasználónak pusztán definiálnia kell, hogy milyen üzenetek léteznek, és megadni, hogy azokat fogadni vagy küldeni szeretné. Ennek a konfigurációnak a hatására a LIN stack folyamatosan frissíteni fogja a fogadni kívánt üzenetek adatmemória részét a hálózatról érkező új adatokkal, illetve periodikusan LIN üzenteket fog formázni a küldeni kívánt üzenetek adatmemória részéből, és el fogja küldeni a LIN hálózatra.



7.12 ábra Minta LIN hálózat

Mintapélda a LIN stack működésére

A hálózatunk tartalmaz 1 LIN master-t és 2 LIN slave-et (7.12 ábra). A hálózat működése a következő: a LIN slave 1-es egység egyik nyomógombjának pillanatnyi értékét megjelenítjük a LIN master és a LIN slave 2 egység hétszegmenses kijelzőjén. Ehhez a következő három főprogramra lesz szükségünk. (A nyomógomb- és kijelzőkezelő függvények leírását megtaláljuk a 7.2. fejezetben vagy a megfelelő mitmót API dokumentációban.)

LIN slave 1 programja (a program include része helytakarékosság miatt kimaradt):

```
U8 Buf_GET_SLAVE [4];          // A slave-ünk üzenetének adatterülete
t_frame MESS_GET_SLAVE;        // Az üzenet tulajdonságait leíró struktúra

int main (void) {
  U8 number_of_frame ;         // Hány üzenetet használ a slave

  // A slave üzenetének specifikációja
  MESS_GET_SLAVE.frame_id    = 0x00 ;     // Üzenet ID
  MESS_GET_SLAVE.frame_size  = 4 ;        // Az Üzenet adathossza
  MESS_GET_SLAVE.frame_type = REMOTE_LIN_FRAME_TYPE ;
                                          // Mi küldjük az üzenetet
  MESS_GET_SLAVE.frame_data  = Buf_GET_SLAVE; // Az üzenet adatmezője

  // A Slave felkonfigurálása, hogy használja az adott üzenetet
  number_of_frame = 1;
  my_schedule.frame_message[0] = MESS_GET_SLAVE;
  my_schedule.number_of_frame = number_of_frame;

  dpy_trm_s01__Init();  // A kijelzőkártya iniciálása
  lin_init();           // A LIN slave és annak hardverének iniciálása
  sei();                // Globális interrupt engedélyezés

  /* Főciklus: a nyomógomb állapotának lekérdezése és eltárolása a
  specifikált LIN üzenet első adatmezőjébe. A nyomógomb állapotának
  kijelzése a helyi hétszegmenses kijelzőre. A nyomógomb főciklus által
  folyamatosan frissített állapotát minden master-kérésre a LIN
  protokolcsomag el fogja küldeni a LIN buszon anélkül, hogy ehhez nekünk
  valamit tennünk kellene */
  while(1) {
      Buf_GET_SLAVE[0] = DPY_TRM_S01__BUTTON_1_GET_STATE();
      dpy_trm_s01__7seq_write_number(Buf_GET_SLAVE[0], 0);
        _delay_ms(10);
  }
  return 0;
}
```

LIN slave 2 programja (a program include része helytakarékosság miatt kimaradt):

```
U8 Buf_SET_SLAVE [4];          // A slave-ünk üzenetének adatterülete
t_frame MESS_SET_SLAVE;        // Az üzenet tulajdonságait leíró struktúra

int main (void) {
  U8 number_of_frame ;         // Hány üzenetet használ a slave

  // A slave üzenetének specifikációja
  MESS_SET_SLAVE.frame_id    = 0x00 ;     // Üzenet ID
  MESS_SET_SLAVE.frame_size  = 4 ;        // Az Üzenet adathossza
  MESS_SET_SLAVE.frame_type = STANDART_LIN_FRAME_TYPE;
                  // Mi kapjuk az üzenetet (STANDART T-vel, nem elírás!)
  MESS_SET_SLAVE.frame_data  = Buf_SET_SLAVE; // Az üzenet adatmezője

  // A Slave felkonfigurálása, hogy használja az adott üzenetet
  number_of_frame = 1;
  my_schedule.frame_message[0] = MESS_SET_SLAVE;
  my_schedule.number_of_frame = number_of_frame;

  dpy_trm_s01__Init();  // A kijelzőkártya iniciálása
  lin_init();           // A LIN slave és annak hardverének iniciálása
  sei();                // Globális interrupt engedélyezés

  /* Főciklus: a specifikált LIN üzenet adatmezőjének az értékét
  folyamatosan kijelezzük a hétszegmenses kijelzőre. Az adatmező értéke
  minden a hálózaton hibátlanul elküldött LIN üzenet hatására frissül,
  anélkül, hogy erre nekünk külön figyelmet kellene fordítanunk */
  while(1) {
        dpy_trm_s01__7seq_write_number(Buf_SET_SLAVE[0], 0);
        _delay_ms(10);  }
  return 0;
}
```

LIN master programja (a program include része helytakarékosság miatt kimaradt):

```
U8 Buf_GET_SLAVE [4];          // A slave-ünk üzenetének adatterülete
t_frame MESS_GET_SLAVE;        // Az üzenet tulajdonságait leíró struktúra

int main (void) {
  U8 number_of_frame ;         // Hány üzenetet használ a slave

  // A slave üzenetének specifikációja
  MESS_GET_SLAVE.frame_id    = 0x00 ;     // Üzenet ID
  MESS_GET_SLAVE.frame_size  = 4 ;        // Az Üzenet adathossza
  // Az üzenet fejlécét elküldjük, de az adatokat egy slave szolgáltatja
  MESS_GET_SLAVE.frame_type = REMOTE_LIN_FRAME_TYPE ;
  /* A master a következő LIN üzenet elküldése előtt ennyit fog várni (a
  megadott érték bitidőben van számolva, aminek az alapja a LIN
  kommunikációs sebessége. Ez jelenleg 19200 baud, vagyis 0,1 másodperces
  ütemezést jelent) */
  MESS_GET_SLAVE.frame_delay  = 1920;
  MESS_GET_SLAVE.frame_data  = Buf_GET_SLAVE;  // Az üzenet adatmezője

  // A master felkonfigurálása, hogy használja az adott üzenetet
  number_of_frame = 1;
  my_schedule.frame_message[0] = MESS_GET_SLAVE;
  my_schedule.number_of_frame = number_of_frame;

  dpy_trm_s01__Init();  // A kijelzőkártya iniciálása
  lin_init();           // A LIN master és annak hardverének iniciálása
  sei();                // Globális interrupt engedélyezés

  /*  Főciklus:  a  specifikált  LIN  üzenet  adatmezőjének  az  értékét
  folyamatosan kijelezzük a hétszegmenses kijelzőre. A háttérben a LIN
  protokoll stack 0,1 másodpercenként elküldi a 0x00 ID-jű üzenet fejlécét
  a buszra, erre a fejlécre válaszként a LIN slave 1 elküldi az üzenethez
  tartozó adatmezőket, amit a busz összes egysége vehet, ha akar */
  while(1) {
       dpy_trm_s01__7seq_write_number(Buf_SET_SLAVE[0], 0);
        _delay_ms(10);
  }
  return 0;
}
```

Mint látható, a felhasználó szempontjából a LIN protokollcsomag működése igen egyszerű: meg kell adni, hogy milyen üzeneteket szeretnénk, azok küldözgetéséről pedig a protokollcsomag automatikusan gondoskodik a háttérben.

—•—