Autóipari beágyazott rendszerek

Dr. Fodor, Dénes

Speiser, Ferenc

Szerzői jog © 2014 Pannon Egyetem

A tananyag a **TÁMOP-4.1.2.A/1-11/1-2011-0042** azonosító számú „ ***Mechatronikai mérnök MSc tananyagfejlesztés*** ” projekt keretében készült. A tananyagfejlesztés az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával valósult meg.

Nemzeti Fejlesztési Ügynökség
www.ujszechenyiterv.gov.hu
06 40 638 638

A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Kézirat lezárva: 2014 február

Lektorálta: Triebl László

Közreműködő: Enisz Krisztián

A kiadásért felel a(z): Pannon Egyetem

Felelős szerkesztő: Pannon Egyetem

2014

Autóipari beágyazott rendszerek

Dr. Fodor Dénes

Tartalomjegyzék

1 Bevezetés ... 7

2 Beágyazott rendszerek definíciója, követelmények ... 8

2.1 Központi vezérlőegység fő típusai ... 9

2.1.1 ASIC (Application Specific Integrated Circuits) ... 9

2.1.2 ASIP (Application-Specific Instruction-set Processor) és DSP (Digital Signal Processor) ... 9

2.1.3 CPLD (Complex Programmable Logic Device) ... 9

2.1.4 FPGA (Field Programmable Gate Array) ... 10

2.1.5 SoC (System On a Chip) ... 11

2.1.6 Mikrokontrollerek ... 12

2.2 Mikrokontrollerek alapvető felépítése ... 12

2.2.1 Memóriák ... 14

2.2.2 Architektúrák ... 15

3 Erőforrás-allokáció, szinkronizáció ... 17

3.1 Memória- és háttértárkezelés ... 17

3.1.1 Memóriakezelés beágyazott rendszerekben ... 17

3.1.2 Háttértárak kezelése ... 25

4 Kommunikáció ... 32

4.1 Az autóiparban leggyakrabban alkalmazott kommunikációs protokollok ... 32

4.1.1 UART és RS-232 ... 32

4.1.2 SPI ... 35

4.1.3 I^2C ... 37

4.1.4 CAN ... 39

4.1.5 CANOpen ... 45

4.1.6 LIN ... 50

4.1.7 MOST ... 54

4.1.8 FlexRay ... 57

5 Beágyazott rendszerek a járműiparban (Biztonságkritikus rendszerek) ... 69

5.1 Az autóipari beágyazott rendszerek szoftverfejlesztésének folyamata. ... 69

5.1.1 Valós idejű rendszerek követelményanalízise, modellezése és modellezési eszközei, HIL (Hardware in the Loop) és a SIL (Software in the loop) típusú szimulációk ... 70

6 Időkezelés és adatátvitel ... 72

Lehetetlenségi tétel (Two Generals' Problem) ... 72

Bizánci tábornokok problémája (Byzantine generals problem) ... 74

Ütemezés ... 76

6.1.1 Task tulajdonságai és az ütemezés kapcsolata ... 76

6.1.2 Ütemezők típusai ... 76

6.2 Task-ok közötti kommunikáció ... 84

7 Szoftverfejlesztési szabványok ... 86

7.1 CMMI modell ... 86

7.1.1 Folyamatközpontú szemlélet ... 86

7.1.2 CMMI modellértelmezések ... 87

7.1.3 Fejlettségi szintek ... 88

7.1.4 Folyamatterületek ... 89

7.2 V-modell ... 90

8 MISRA-C ... 93

8.1 Hogy néznek ki a MISRA szabályok? ... 93

8.2 Statikus kódelemzés ... 94

9 Tesztfeladatok ... 96

10 Irodalomjegyzék ... 97

Ábrajegyzék

2.1. ábra A beágyazott rendszerek általános felépítése 8

2.2. ábra CPLD elvi felépítése 10

2.3. ábra FPGA elvi felépítése 11

2.4. ábra A mikrokontroller alapvető részei 12

2.5. ábra Neumann- és Harvard architektúra 16

2.6. ábra Utasítás kezelés a Neumann- és Harvard architektúra esetén 16

3.1. ábra A veremhez tartozó mintaprogram memóriaábrája 18

3.2. ábra A veremhez tartozó mintaprogram memória ábrája 20

3.3. ábra A halomhoz tartozó első mintaprogram memóriaábrája 22

3.4. ábra A halomhoz tartozó második mintaprogram memóriaábrája 24

3.5. ábra Hagyományos mágneses háttértár felépítése 30

3.6. ábra Lebegő kapus MOSFET, a Flash memóriák alapegysége 30

4.1. ábra UART/RS232 protokoll által definiált üzenetformátum 34

4.2. ábra Az SPI hálózat elrendezése két szolga esetén 36

4.3. ábra Az SPI kommunikáció elve 36

4.4. ábra Egy példa az I^2C hálózat elrendezésére 37

4.5. ábra Egy példa az I^2C kommunikációra 39

4.6. ábra Példa egy nagysebességű CAN hálózat felépítésére 3 csomópont esetén 40

4.7. ábra Példa egy alacsony CAN hálózat felépítésére 3 csomópont esetén 40

4.8. ábra Az adat típusú CAN üzenetek felépítése 40

4.9. ábra CAN csomópont hibaállapotok közötti átmenete 43

4.10. ábra A CAN csomópontok működési diagramja 45

4.11. ábra Az alkalmazási réteggel kibővített CAN modell 46

4.12. ábra A Mester-szolga kommunikációs modell 47

4.13. ábra A Kliens-szerver kommunikációs modell 47

4.14. ábra A gyártó-fogyasztó kommunikációs modell 48

4.15. ábra A LIN üzenetek fő részei 50

4.16. ábra Hagyományos ellenőrző összeg számítása két adatbájt esetén 51

4.17. ábra Kiterjesztett ellenőrző összeg számítása két adatbájt esetén 52

4.18. ábra A MOST buszrendszer működése 54

4.19. ábra A MOST 25 esetében alkalmazott átviteli protokoll 56
4.20. ábra A MOST 50 esetében alkalmazott átviteli protokoll 56
4.21. ábra A MOST 150 esetében alkalmazott átviteli protokoll 57
4.22. ábra Egy FlexRay hibrid topológiájú hálózat felépítése 59
4.23. ábra A busz meghajtó belső logikai felépítése 60
4.24. ábra FlexRay kommunikációs ciklus szerkezete 62
4.25. ábra: Egy statikus szegmens felépítése 63
4.26. ábra: Egy dinamikus szegmens felépítése 63
4.27. ábra: Egy szimbólum ablak felépítése 63
4.28. ábra: FlexRay keret 65
4.29. ábra: A FlexRay többségi szavazás algoritmusának a szemléltetése 66
4.30. ábra: A FlexRay óra szinkronizáció mente 66
4.31. ábra A macro- és microtickek felépítése 67
6.1. ábra Két tábornok problémája 73
6.2. ábra Két tábornok problémája animáción bemutatva 73
6.3. ábra Tábornok 3 az áruló 74
6.4. ábra Tábornok 1 az áruló 75
6.5. ábra Task futásának fontosabb időpontjai 76
6.6. ábra Folyamat állapotai ütemezés szempontjából 78
6.7. ábra Megszakításos ütemező 78
6.8. ábra Nem megszakításos ütemező 79
6.9. ábra Egyszerű ütemezők típusai 80
6.10. ábra Prioritásos ütemezők típusai 82
7.1. ábra A CMMI által definiált szintek 89
7.2. ábra A V-modell 91
8.1. ábra A statikus kódelemző célja 94

1 Bevezetés

Napjaink járművei egyre összetettebbé, egyre bonyolultabbá válnak annak érdekében, hogy kielégítsék az egyre növekvő biztonsági és kényelmi követelményeket. A legtöbb ilyen funkciót manapság már beágyazott számítógépek vezérlik kezdve a biztonságkritikus rendszerektől, mint például az elektronikus menetstabilizáló rendszerek, a kényelmi szolgáltatásokig, mint például a légkondicionálás szabályozása.

A modern gépjárművekben közel 40 – 50 darab, egy luxus kategóriás gépjárműben pedig még ennél is több ilyen beágyazott számítógép más néven elektronikus vezérlőegység (ECU) található. Ezek az egészen egyszerű, néhány száz soros programkódot futtató, 8 bites kontrollerektől kezdve az asztali számítógépek teljesítményével összemérhető, modern operációs rendszereket futtató fedélzeti számítógépekig változhatnak.

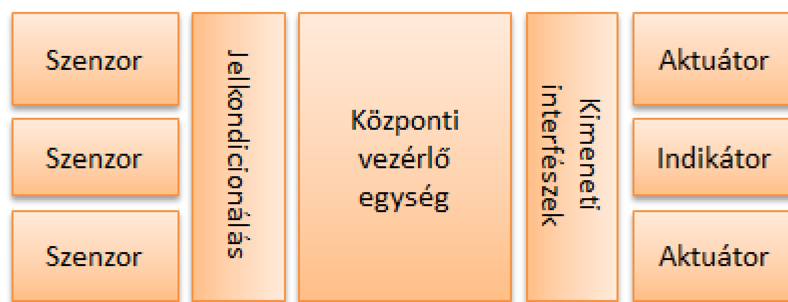
Az eltérő funkciókat ellátó rendszereknek eltérő követelményeknek kell megfelelniük, mind stabilitás, mind megbízhatóság szempontjából. Ilyen követelmény lehet például a elektronikus fékrendszerek (EBS, Electronic Break Systems) esetében, hogy a számításokat illetve a vezérlési feladatokat két külön kontrollernek kell futtatnia párhuzamosan azért, hogy ellenőrizni tudják egymás eredményeit, ezzel lehetőleg kizárva a hibás működésből eredő téves beavatkozások lehetőségét. Ugyanakkor ilyen megkötésekre a légkondicionáló szabályozását végző vezérlőegységnél már nincsen szükség, mivel az nem biztonságkritikus rendszer.

Ahhoz, hogy a mérnökök megfelelően fel tudjanak készülni a jövőben ellátandó tervezési feladatokra, lényeges megismerniük azt, hogy mely rendszereket lehet beágyazott rendszereknek nevezni, ezeknek milyen fajtái vannak illetve milyen tervezési és fejlesztési eljárások terjedtek el az autóiparban.

2 Beágyazott rendszerek definíciója, követelmények

A beágyazott rendszereket számos eltérő felépítésben és eltérő célokra szoktak alkalmazni. Egy adott feladatot ellátó kis számítógépet akkor neveznek beágyazott rendszernek (angolul: embedded system), ha cél-specifikusan lett megtervezve, azaz egy adott jól ismert feladat megoldására illetve ellátására lett kialakítva. Az általános célú számítógépekkel szemben – mint például a személyi számítógép – egy beágyazott rendszer csupán néhány, előre meghatározott feladatot képes ellátni, illetve sokszor tartalmaz feladat-specifikus mechanikus és elektronikus alkatrészeket.

Ezen eszközök kialakítása és tulajdonságai a végrehajtandó feladat függvényében széles skálán mozoghatnak, ugyanakkor alapvető felépítésük többnyire követi a 2.1. ábra által szemléltetett architektúrát.



2.1. ábra A beágyazott rendszerek általános felépítése

Ezen cél-specifikus rendszerek szenzorokon, indikátorokon és aktuátorokon keresztül tartják a kapcsolatot a környezetükkel, illetve ha nem autonóm rendszerként működnek, akkor többnyire valamilyen szabványos kommunikációs interfésszel is rendelkeznek, mint például az SPI, IIC (I^2C), U(S)ART, USB, CAN, FlexRay, Ethernet, WiFi, Bluetooth, ISM rádió, ZigBee, stb.

Sokszor tévesen szokták állítani, hogy egy rendszer a központi egységtől lesz beágyazott rendszer, ugyanakkor sokkal inkább a hardver – szoftver – felhasználás hármas határozza meg, hogy egy adott rendszer beágyazott rendszernek minősül-e. Több felhasználási területen összemosódnak a határok az általános célú és beágyazott rendszerek között. Például egy ipari kisméretű számítógép alkalmazható teljes értékű asztali számítógépként és beágyazott rendszerként is a műszaki környezetétől és a rajta futó programoktól függően.

Jóllehet számtalan különböző felépítésű beágyazott rendszer létezik akár teljesen eltérő architektúrával, ugyanakkor mindegyikben közös, hogy a rendszer feladatai a tervezés idején is egyértelműen specifikálva vannak, így a tervezők a feladatnak megfelelően tudják optimalizálni a rendszert. Ennek köszönhetően az adott feladathoz igazítható a rendszer mind hardver, mind szoftver szempontjából, így csökkenthetőek a költségek és méret is, illetve növelni lehet a megbízhatóságot. Az ilyen rendszerek gyakran egyszerűnek látszanak, azonban a tervezésük gyakran több fajta ismeret szintézisét igényli, mint például a hardveres ismeretek, az alkalmazandó kommunikációs szabványok ismerete, hardver közeli, beágyazott szoftverfejlesztés, PC-s eszközillesztők és magas szintű szoftver ismeretek (PC-s felhasználói felület fejlesztéséhez), intelligens algoritmusok ismerete nagy bonyolultságú feladatok

megoldásához, jelfeldolgozási ismeretek és egyéb alkalmazás-specifikus ismeretek (pl. képfeldolgozás, motorvezérlés).

2.1 Központi vezérlőegység fő típusai

A központi vezérlőegységek a beágyazott rendszerek fő komponensei. Egy adott rendszer több - akár eltérő típusú - vezérlőegységet is tartalmazhat, az ellátandó feladattól függően. A speciális funkció sok esetben ellátható kis számítási teljesítménnyel is, ilyenkor jellemzően a rendszer fogyasztása és költségei egyaránt alacsonyak. A központi vezérlőegység lehet akár mikrokontroller vagy nagy komplexitású vezérlő logika (FPGA) illetve a hagyományos számítógépeknél alkalmazott processzor is.

2.1.1 ASIC (Application Specific Integrated Circuits)

Alkalmazás-specifikus integrált áramkört mindig egy adott specifikus feladat ellátására tervezik, vagyis nem általános felhasználásra. Megtervezése és a fejlesztői szériák legyártása nagy költségekkel járhat bonyolult feladatok megoldása esetén, mivel csak egy adott célra lehet alkalmazni. Ugyanakkor nagy tételben történő felhasználás esetén költséghatékony, valamint olyan egyedi feladatoknál, amelyek más eszközzel nem oldhatóak meg.

2.1.2 ASIP (Application-Specific Instruction-set Processor) és DSP (Digital Signal Processor)

Az alkalmazás orientált processzorok (ASIP) és a digitális jelfeldolgozó processzorok (DSP) egy osztályba sorolhatóak (a DSP egy ASIP). Jellegzetességük, hogy utasításkészletük egy adott célfeladathoz lett optimalizálva és a célfeladat ellátásához legszükségesebb utasításokat tartalmazzák. A jelfeldolgozáshoz optimalizált processzorok Harvard-architektúrát alkalmaznak és gyors, speciális feladatú hardver szorzó-akkumulátor modullal rendelkeznek (MAC egységek (Multiply-Accumulate unit)), melynek segítségével több műveletet egy lépésben képesek elvégezni (pl.: $a \leftarrow a + (b \times c)$), valamint egy órajel alatt több memóriacím elérésre is képesek, így gyorsítva fel a műveletvégzést, melyet a processzorral egybeintegrált gyors memória és/vagy gyorsítótár is elősegít. Fontos megjegyezni, hogy ezek a speciális központi egységek sokszor nem önállóan, hanem társprocesszorként jelennek meg egy mikrokontroller vagy más központi egység mellett, gyakran azzal egy tokba integrálva.

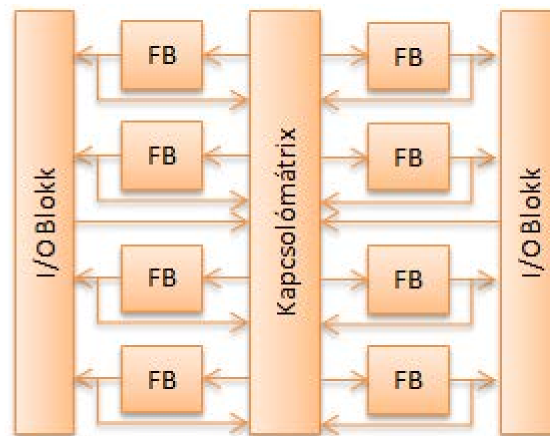
2.1.3 CPLD (Complex Programmable Logic Device)

A CPLD vagy más néven összetett programozható logikai áramkör lényegében több, egyszerű programozható logikai egység egybeintegrálása oly módon, hogy ezen egységek kimenetei és bemenetei összekapcsolhatóak egymással.

A programozható logikai egységek (PLD) lényegében olyan logikai kapuk, flip-flop-ok stb. együttese, melyet a felhasználó tud konfigurálni, így hozva létre különböző logikai kapcsolásokat. Ezeket a logikai egységeket a CPLD-k esetében általában makrocelláknak hívják és a funkcionális blokkokon belül helyezkednek el (többnyire 4-16 ilyen egység található egy funkcionális blokkban) (2.2. ábra).

A logikai hálózat a funkció blokkok programozásával hozható létre illetve egy kapcsolómátrix segítségével lehet összekötni a funkció blokkok ki- és bemeneteit, valamint a tokozás ki- és bemeneteit, azaz az I/O blokkokat.

A CPLD-k előnyös tulajdonsága, hogy az áramkör a nyomtatott áramköri panelra történő beültetés után is programozható, illetve újraprogramozható, valamint a kapcsolatok és a makrocellák konfigurációját, azaz jelen esetben a programot flash típusú memóriában tárolja, így kikapcsolás után is megőrzi tartalmát.



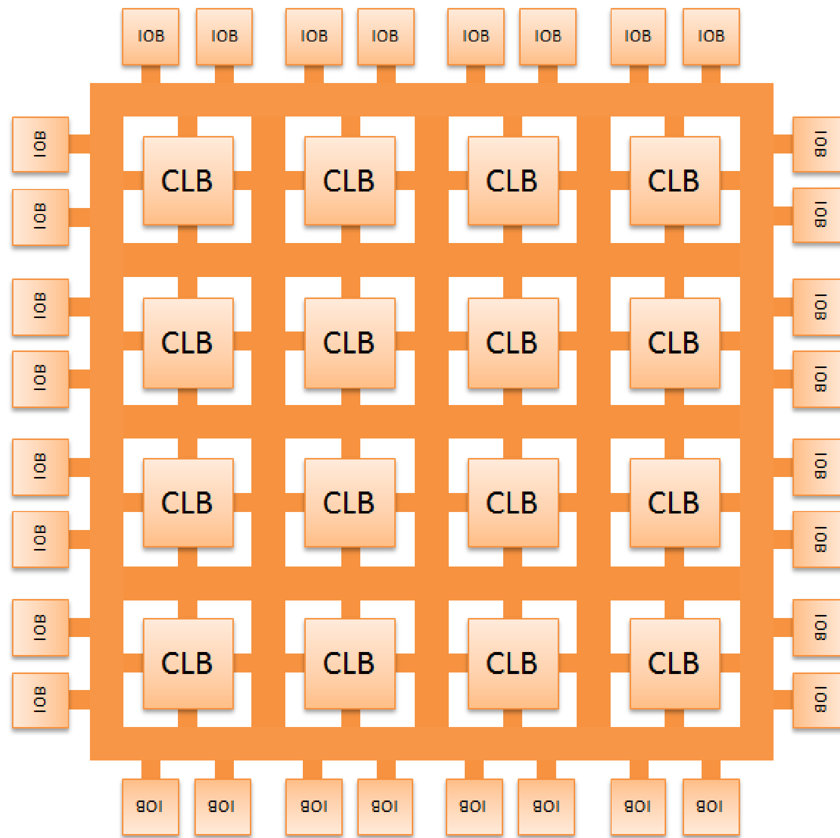
2.2. ábra CPLD elvi felépítése

2.1.4 FPGA (Field Programmable Gate Array)

Az FPGA vagy más néven programozható kapu mezők lényegében a felhasználó által programozható kapu-áramkörök. Logikai hálózat kialakításánál a konfigurálható logikai blokkokat (CLB) és az ezek közötti összeköttetéseket kell programozni. Ezek a blokkok belső huzalozási utak felhasználásával tetszőlegesen összeköthetők egymással és lényegében a konfigurálható logikai blokkok valósítják meg a felhasználónak szükséges logikai kapcsolatokat (2.3. ábra).

A programozható ki- és bemeneti blokkok, azaz az IOB-k teremtik meg a kapcsolatot a tokozás kivezetései és a belső logikai kapcsolás között. Általában mindegyik IOB definiálható bemenet vagy kimenet illetve kétirányú csatlakozásként is.

A programozható kötések segítségével egymáshoz kapcsolhatóak a konfigurálható logikai blokkok valamint a ki- és bemeneti blokkok kivezetései, az összeköttetések állapotait pedig egy konfigurációs memória tárolja.



2.3. ábra FPGA elvi felépítése

Az FPGA logikai erőforrásaiból eredő párhuzamosság jelentős számítási teljesítményt tesz lehetővé, ugyanakkor jellemzője a hosszú fordítási idő.

Sokszor össze szokták keverni a CPLD és FPGA alapú központi egységeket, holott többnyire ezek architektúrája (ahogy a korábbi ábrákon is látható) valamint felhasználási körük is jelentősen eltér. Általánosságban elmondható, kicsit egyszerűsítve a dolgot, hogy az FPGA adott számú kaput tartalmaz, amiket rugalmasan lehet felhasználni adott blokkok kialakítására. A CPLD esetében ezen blokkok fixek (ezek a makrocellák), ezeken belül lehet logikai kapcsolásokat létrehozni, majd a makrocellákat lehet összekötni. További fontos felépítésbeli különbség, hogy az FPGA SRAM alapú, így kikapcsolás után újra fel kell rá tölteni a programot egy külső, nem felejtő memóriából, míg CPLD többnyire flash alapú memóriában tárolja a programot, így az kikapcsolás után is megőrződik, ezért nem kell visszatölteni az induláskor.

A felhasználás területén ökölszabályként elmondható, hogy az egyszerűbb, gyorsabb válaszidőt igénylő feladatok esetén többnyire CPLD-t, míg az összetettebb, több feladatszálat futtató feladatok esetén FPGA-t szoktak alkalmazni.

2.1.5 SoC (System On a Chip)

Olyan integrált áramkörök, melyeknek részét képezik a perifériakezelő rendszerek, a CPU-mag(ok), jellemzően az integrált grafikus vezérlő. Definíció szerint ezen rendszereket csak egy hajszál választja el a mikrokontrollerektől, ugyanakkor jellemzőjük a nagyobb teljesítmény, nagyobb memória méret, stb. Gyakran x86 (x64) PowerPC vagy ARM

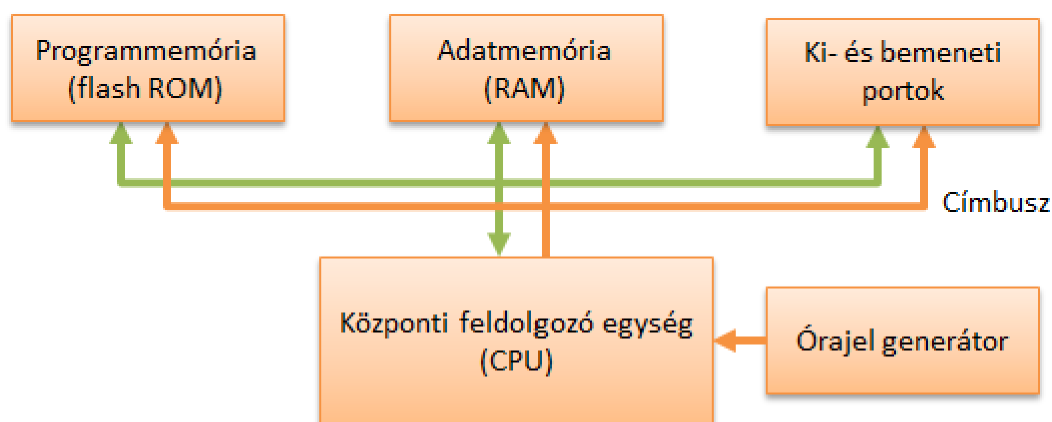
architektúrájú rendszerek, melyek nagyobb teljesítményűek és általánosabb felhasználásúak, mint a mikrokontrollerek.

2.1.6 Mikrokontrollerek

A mikrokontrollerek egychipes áramkörök, melyek egy mikroszámítógép konfiguráció minden elemét (CPU, memória (RAM, ROM), I/O egységek, rendszer órajel generátor, stb.) tartalmazza. Túlnyomórészt Harvard architektúrát és csökkentett utasításkészletet tartalmaznak (Reduced Instruction Set Computing, RISC), valamint külső memóriával általában nem vagy csak az I/O vonalak felhasználásával bővíthetőek. Általában jellegzetes erőforrásokkal rendelkeznek például Watchdog timer, külső és belső megszakítás (interrupt) vonalak, számláló/időzítő áramkörök, digitális és analóg be- és kimeneti vonalak. A kivezetések számának csökkentése céljából többcélú kivezetéseket használnak, azaz egy kivezetéshez több funkció van rendelve. Általában többféle hardveresen integrált kommunikációs interfésztmodult tartalmaznak (pl. UART, SPI, CAN, USB, Ethernet).

2.2 Mikrokontrollerek alapvető felépítése

A beágyazott rendszerekben a legelterjedtebb vezérlőegység típus a mikrokontroller, amelynek tükrében érdemes jobban megismerni a központi vezérlő és a hozzá tartozó kiegészítő elemek főbb típusait.



2.4. ábra A mikrokontroller alapvető részei

Egy mikrokontroller alapvetően a 2.4. ábra által szemléltetett fő komponensekből épül fel:

- Központi feldolgozó egység (CPU):
- A processzor működésének vezérlését és a feladatok végrehajtásának ütemezését a vezérlő egység (CU vagyis a Control Unit) végzi
- Az aritmetikai-logikai egység (ALU) felelős a számítások illetve műveletek végrehajtásáért. Ezt gyakran ki szokta egészíteni egy FPU, azaz lebegőpontos egység, mely a lebegőpontos számokon történő számítások elvégzését hivatott felgyorsítani.
- Alapvető regiszterek a műveletek végrehajtásához, ilyen például a programszámláló, a verem mutató, valamint a státusz regiszter.
- Az átmeneti eredmények tárolására szolgáló, igen gyors elérésű regiszterek.
- Utasítás értelmező és egyéb részegységek, melyek a központi feldolgozó egység vezérléséért illetve a megszakítások kezeléséért felelnek.

- Egy kontroller több magot is tartalmazhat, melyek önállóan, egymással párhuzamosan képesek utasításokat végrehajtani. Ennek igazán a műveletek párhuzamosításánál van jelentősége.
- A végrehajtandó programot tároló memória a programmemória, amely egy nem felejtő, alapesetben csak olvasható (read-only memory (ROM)) memória terület, azaz kikapcsolás után is megőrzi a tartalmát és a mikrokontroller hagyományos működése közben sem íródhat felül.
- Az adatok tárolására szolgáló memória az adatmemória. Ez egy tetszőleges hozzáférésű memória (random access memory (RAM)), azaz működés közben bármely valós memóriacímén írható és olvasható.
- A be- és kimeneti portok (más néven a periféria (I/O) egység) a külvilággal történő kommunikációra szolgálnak.
- A processzor a vezérlőbuszon keresztül utasítja a többi elemet a megfelelő működésre (kétirányú), a címbusz a memória (vagy a periféria) megfelelő tároló rekeszét címzi (egyirányú), míg az adatbuszon mozognak a különféle adatok (kétirányú).
- Az órajel generátor felel azért, hogy a mikrokontroller összes komponense összhangban legyen. Maga az órajel származhat a mikrokontrollerbe integrált órajel generátortól, illetve egy külső órajel generátortól is. Sokszor a kontrollerbe építve, processzorokban található egy kvarckristály, ami a működéshez szükséges órajelet szolgáltatja. A processzor részegységei az órajel ütemére végzik feladataikat; amikor egy részegység megkapja az órajelet egy elektronikus jel formájában, akkor elvégzi a soron következő műveletet, majd amikor megkapja a következő jelet, akkor a következő műveletet végzi el. A műveletet nem szabad összetéveszteni az utasítással: egy utasítás végrehajtása több órajel ciklust is igénybe vehet. Az órajel fontos jellemzője a processzornak, de nem jellemzi egyértelműen a teljesítményét, mivel sok processzor egy órajel alatt több műveletet is el tud végezni, mely tulajdonságot az IPC (Instructions Per Cycle) értékkel, azaz az egy órajel ciklus alatt elvégzett műveletek számával lehet jellemezni.

A kontrollerekben általánosan megtalálhatóak az imént felsorolt komponensek, ugyanakkor a perifériák terén már nem ilyen nagy az összhang. A perifériák kezeléséért többnyire különálló komponensek felelnek, melyeket kontrollerbe integrálnak. Ezek kialakítása és szerepe eltérő lehet a különböző kontrollerek esetén, ugyanakkor elmondható, hogy vannak olyan perifériák, amelyek a kontrollerek többségében megtalálhatóak:

- A beépített időzítőből többnyire legalább egy megtalálható a mikrokontrollerekben. Ezeknek számos feladata lehet például használható időmérésre vagy különböző feladatok ütemezésére.
- Biztonsági időzítő áramkör, azaz más néven WatchDog Timer (WDT), melynek a feladata, hogy újraindítsa a kontrollert abban az esetben, ha az végtelen ciklusba kerülne valamely műveletnél.
- A valósidejű órajel (RealTime Clock, RTC) generátor feladata a hosszútávon történő pontos időmérés. Gyakran külső elemes vagy akkumulátoros tápellátás is szükséges a működtetéséhez, hogy abban az esetben is tudja mérni az időt, amikor a mikrokontroller kikapcsolt állapotban van.

- Adatok tárolására szolgáló, nem felejtő memóriaterület (pl.: azonosítók, hálózati címek tárolására).
- Analóg-digitális átalakító (ADC). A feladata, hogy a beérkező analóg jelet mintavételezés és kvantálás után, a központi feldolgozó egység által értelmezhető digitális formára alakítsa.
- Digitális-analóg átalakító (DAC). A feladata, hogy a digitális jeleket analóg jellé alakítsa.
- Kommunikációs interfészek a külvilággal illetve más mikrokontrollerekkel történő kommunikációt tesznek lehetővé. Ilyen lehet például az UART, I^2C, SPI, stb.
- A fejlesztés során alkalmazott komponenseknek az a fő feladatuk, hogy a fejlesztési stádiumban elősegítsék a mikrokontroller felprogramozását, valamint a programban történő hibakeresését (debugger).

A perifériák kezelése többnyire meghatározott memóriaterületek írásával és olvasásával történik (Special Function Register, SFR), így az alapvető felépítésben nem igényel különösebb változtatásokat.

2.2.1 Memóriák

A memóriáknak alapvetően két fajtája van, amelyek közül az egyik a CPU-ba került integrálásra, melyet regiszternek hívnak. A másik, amit a CPU buszrendszeren keresztül ér el, hagyományosan memóriának szokás hívni. A memória lehet alapesetben csak olvasható (ROM), valamint írható és olvasható is (RAM). Ezen felül egy gyakori csoportosítás még a felejtő és nem felejtő memóriák osztálya:

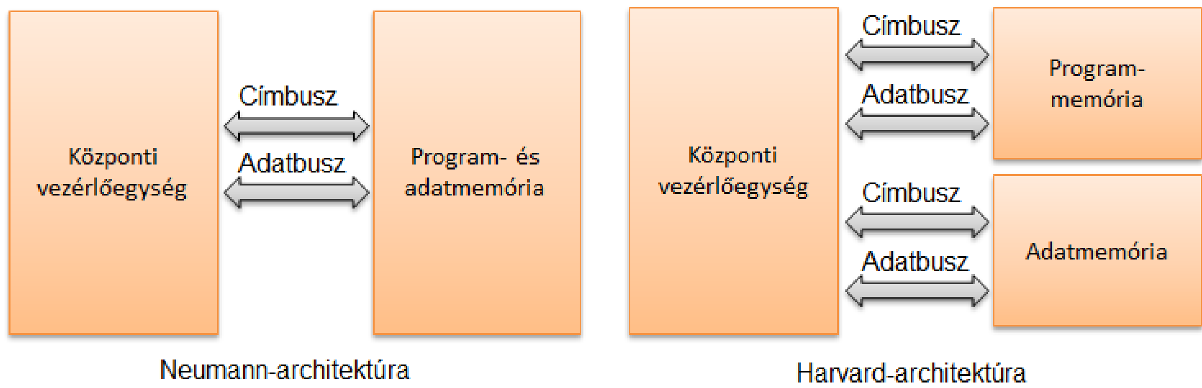
- A felejtő memóriákat gyakran félrevezetően RAM-nak szokták nevezni. A felejtő memória lényege, hogy írni és olvasni is lehet, ugyanakkor, ha megszűnik a kontroller tápellátása, akkor elveszíti a tartalmát. A kontrollerek többsége kevés felejtő memóriát szokott tartalmazni nagy helyigénye és magas ára miatt. Mivel nem a számítógépeknél alkalmazott dinamikus RAM-ot (DRAM), hanem statikus RAM-ot (SRAM) szoktak használni, azaz csak akkor kell frissíteni a tartalmát, amikor változik, ezért nem igényel folyamatos újraírást, mint ahogy az a dinamikus RAM esetében történik.
- A nem felejtő memóriát gyakran félrevezetően ROM-nak szokták nevezni. Előnyük, hogy tartalmukat a tápellátás megszűnésekor is megőrzik valamint, hogy a modern kontrollerek többsége írni is képes a nem felejtő memória területet. Ugyanakkor hátránya, hogy az írási művelet sokszor jóval lassabb, mint a felejtő memória esetében. A nem felejtő memóriáknak számos fajtája van:
- A maszkolt ROM ténylegesen csak egyszer írható, ugyanis a tartalmát például fotólitográfiás eljárással a gyártás során ténylegesen beleégetik a memóriába. Első sorban nagy szériás gyártásban gazdaságos.
- Az EPROM (erasable programmable read only memory), azaz az elektronikusan programozható ROM, ahogy a neve is mutatja elektronikusan programozható, ugyanakkor törölni elektronikus úton nem lehet, csak például ultraibolya fénnyel. A Flash memóriák elterjedése előtt széles körben alkalmazták.
- Az OTP (one-time programable memory), más néven egyszer írható ROM, lényegében egy EPROM, amelyet nem lehet törölni.

- Az EEPROM elektronikusan törölhető és programozható ROM hasonló az EPROM-hoz, ugyanakkor törlése történhet elektromosan is.
- Flash memória. Leginkább az EPROM-hoz hasonlít, mivel írni és törölni is lehet elektromosan, ugyanakkor mégsem ugyanaz a kettő. A fő különbség a felépítésbeli eltéréseken túl, hogy az EPROM-ot bájtonként, míg a NAND Flash memóriát blokkonként lehet törölni (a NOR flash bájtonként is címezhető). Fontos megjegyezni, hogy mind az EPROM, EEPROM, valamint a Flash memória esetében a memóriacelláknak van egy elhasználódási száma. Azaz nem lehet végtelenszer törölni majd újraírni őket, mivel fizikai működési elvükből adódóan egy idő után „elhasználódnak". Szerencsére a gyártók nagy hangsúlyt fektettek az élettartam növelésére, elsősorban a Flash memóriáknál, így a törlési/írási ciklusok száma a milliós vagy még magasabb nagyságrendet közelíti, így egyre kevésbé jelent problémát.
- Az FRAM-nak (Ferroelectric Random Access Memory) a hagyományos, nem felejtő memóriákkal szemben több előnye is van. Felépítése leginkább a dinamikus memóriákéhoz hasonlít, ugyanakkor a dielektromos réteg helyett ferroelektromos réteget alkalmaz a memóriacelláknál. Ezzel lényegében kiküszöböli annak állandó frissítési igényét (vagyis, hogy még akkor is folyamatosan frissíteni kelljen benne az adatot, ha van tápfeszültség), sőt még nem felejtővé is teszi, azaz a tápellátás megszűnése után is megőrzi tartalmát. FRAM-ok esetében sokszor kisebb az elhasználódás mértéke, mint a Flash memóriáknál (ez elsősorban az alacsony feszültségen működő FRAM-okra igaz), valamint az SRAM-okhoz hasonló írási idővel rendelkeznek (100 ns környéki vagy még kisebb), így jelentősen gyorsabbak, mint a Flash memóriák. A fenti előnyök mellett adott lenne, hogy az FRAM helyettesítse az igen elterjedt Flash memóriákat, ugyanakkor magasabb ára és nagyobb fizikai mérete ezt nem mindig teszi lehetővé.

2.2.2 Architektúrák

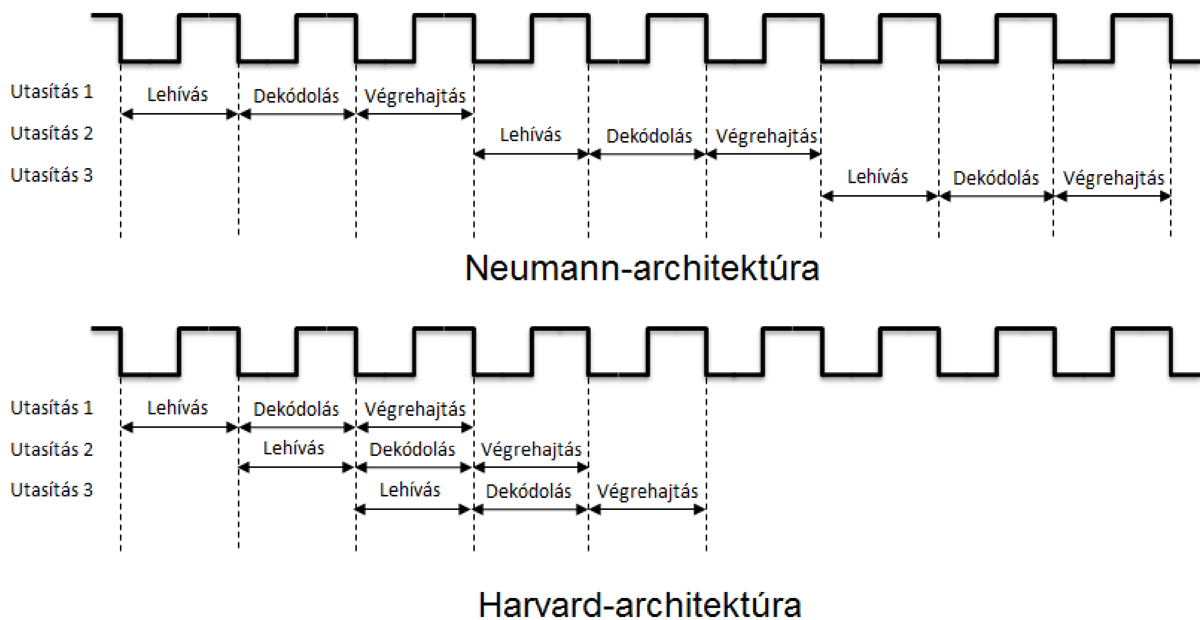
A beágyazott rendszerekben alkalmazott kontrollerek többnyire két különböző architektúrát alkalmaznak a memóriaelérés, valamint memória felépítésének szempontjából. Ez a két architektúra a Harvard- illetve a Neumann-architektúra. Mindkét felépítés esetén a fő részegységek funkciója megegyezik.

A Neumann-architektúra esetén az adat- (nem felejtő) és a programmemória (felejtő) fizikailag nem választhatóak szét. További jellemzője a szekvenciális utasítás végrehajtás. A Harvard-architektúránál a program- és adatmemória fizikailag szeparálható (2.5. ábra), továbbá minden utasítás egyszavas, így téve gyorsabbá az utasítás feldolgozását, mivel a rövidebb utasítások értelmezésére kevesebb idő kell. (A programmemória szóhosszúsága tetszőleges lehet, így nem jelent korlátozó tényezőt, valamint az adatmemória általában bájtos kialakítású.)



2.5. ábra Neumann- és Harvard architektúra

A speciális kialakítás miatt egyszerre (párhuzamosan) is kezelhetőek a memóriák, nem alakulhat ki versenyhelyzet az adat és kód elérésénél. Ez jelenti a legfőbb különbséget a két architektúra esetén, mivel mind a két esetben az utasítások feldolgozása három részre osztható: utasítás lehívás, dekódolás és végrehajtás. Ugyanakkor a Harvard-architektúrájú processzorok esetén egyszerre címezhető a program- és az adatmemória, így az első utasítás dekódolásával egyidejűleg lehívható a második utasítás kódja és így tovább (2.6. ábra).



2.6. ábra Utasítás kezelés a Neumann- és Harvard architektúra esetén

A beágyazott rendszerek többségénél alkalmazott vezérlőegységek Harvard-architektúrát használnak.

3 Erőforrás-allokáció, szinkronizáció

A beágyazott rendszereknél kulcsfontosságú, hogy milyen erőforrások érhetőek el, illetve ezekkel hogyan gazdálkodik a rendszer. Ilyen erőforrások lehetnek az operatív memória, a háttértárak valamint maga a processzor is. Az erőforrás-allokáció illetve -kezelés azt határozza meg, hogy a rendszer milyen módon kezeli vagy ütemezi az adott erőforrásokat a feladatoknak megfelelően.

3.1 Memória- és háttértárkezelés

Elsősorban a memóriakezelés lényeges minden beágyazott rendszer esetében, ugyanakkor a nagyobb rendszerekben vagy olyan eszközök esetén, melyek rendelkeznek hagyományos háttértárral, a tárkezelés is fontos.

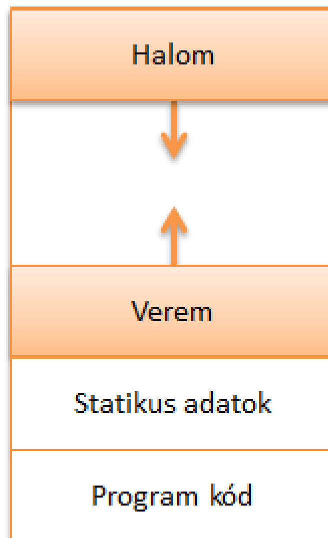
3.1.1 Memóriakezelés beágyazott rendszerekben

A memória az egyik legfontosabb (és gyakran a legszűkösebb) erőforrás, amivel egy beágyazott rendszernek (vagy egy operációs rendszernek) gazdálkodnia kell. Beágyazott rendszerek esetén a memória eleve kicsi, hiszen általában a mikrovezérlők belső RAM-ja áll csak rendelkezésre, és - mint a költséghatékonyság miatt minden más is -, a memória is a lehető legkisebb méretű. A nagy operációs rendszerek esetén főleg a több felhasználós rendszerekben jelentkeznek memóriakezelési problémák, ahol gyakran olyan sok és nagy folyamat fut, hogy együtt nem férnek be egyszerre a memóriába.

A beágyazott rendszerek esetén többnyire három memóriakezelési stratégia terjedt el. A gyakorlatban egy rendszeren belül általában több megoldás is előfordul, a szerint alkalmazva az egyes stratégiákat, hogy milyen előnyös és hátrányos tulajdonságokkal rendelkeznek.

- Statikus memóriafoglalás
- Verem (stack) alapú memóriakezelés
- Halom (heap) alapú memóriakezelés

A kevert memóriakezelést használó program/szál tipikus memóriastruktúráját a 3.1. ábra szemlélteti. Más operációs rendszereknél illetve mikrokontrollerek esetén természetesen ez típustól illetve operációs rendszertől függően változhat. Illetve lehetnek további részek, mint például a környezeti változókat valamint parancssori paramétereket tartalmazó szekció.



3.1. ábra A veremhez tartozó mintaprogram memóriaábrája

Az ábrán legalul található a program kód (text, code), amelyet a statikus adatok követnek, melyek a program indulásakor töltődnek be. Jelen esetben e felett helyezkedik el a verem. A verem és a halom által elfoglalt memóriaterület a szabadon felhasználható memória, melyből mind a halom, mind a verem lefoglalhat. Elsősorban a mikrokontrollerek esetében a verem és a halom maximális mérete kötött szokott lenni, így a szabadon felhasználható memória voltaképpen nem áll rendelkezésre, hanem a halmon és a vermen belüli szabad területről lehet beszélni. Lényeges, hogyha egy program megpróbál kicímezni a rendelkezésére álló memóriaterületből, azaz egy olyan memória területre próbál hivatkozni, ami nem létezik vagy nem hozzá tartozik, az hibát fog kiváltani a program futásakor.

3.1.1.1 Statikus memóriafoglalás

A statikus memóriakezelés esetén a memóriaterületek kiosztása nem futás közben történik, hanem már fordításkor eldől, hogy mi hova fog kerülni a memóriában. Ez azt jelenti, hogy az egyes függvények, feladatok és taszkok előre meghatározott memóriaterületeket kapnak. A memória kiosztása rögzített, és a program futása közben nem változhat meg. Az egyes memóriacímeken lévő adatokat mutatókkal lehet elérni.

A módszer előnye, hogy már fordításkor pontosan tudható, hogy mekkora memóriára lesz szükség futás közben. Emellett számos olyan problémát ki lehet kerülni, mely az egyes memóriaterületek futás közben változó használati módjából ered.

A hátrányai közé tartozik, hogy az újrahívhatóságot igénylő műveletek esetén nem, vagy csak nehézkesen alkalmazható, ilyen például a rekurzió. További hátrány, hogy az egyes függvények/feladatok/szálak számára szükséges memória méretét előre ismerni kell, mely valamelyest ront a korábban említett teljes memóriaigény ismeretére vonatkozó előny mértékén. Ebből adódóan minden memóriaterület, amire futás közben szükség lehet, végig le van foglalva függetlenül attól, hogy épp használatban van-e.

Ugyanakkor a statikus memóriakezelésre alapozva létre lehet hozni olyan adatszerkezeteket, amelyek a felsőbb szoftveres szintekről nézve különböző mértékben dinamikusnak tűnnek. Ilyenek adatszerkezetek például a verem és halom.

3.1.1.2 Verem (stack) alapú memóriakezelés

A verem memória LIFO (Last In First Out) szervezésű, azaz folyamatosan lehet bele adatokat tölteni, de mindig az utolsóként bekerült adatot lehet elérni belőle. A kiolvasott adatok olvasáskor törlődnek a veremből, így onnantól kezdve már az eggyel korábbi adat fog a verem tetején elhelyezkedni. Az alapértelmezett verem-memórián két művelet végezhető el:

- új elem hozzáadása (push),
- az utolsó elem eltávolításával járó kiolvasás (pop).

Ezen felül definiálhatóak további műveletek, mint például a teljes verem ürítése, az utolsó elem megtekintése (ezzel meg lehet nézni, hogy milyen elem található a verem tetején, ugyanakkor nem kerül törlésre a veremből, mint a pop-nál), ugyanakkor ezek a műveletek többnyire az alacsonyabb pop és push műveleteken alapulnak.

A verem tehát egy olyan statikus lefoglalt memóriaterület, melynek blokkjait egy felsőbb szoftverszintről dinamikusan vehetjük használatba. A verem speciális tulajdonsága, hogy a tartalma fel-le változik.

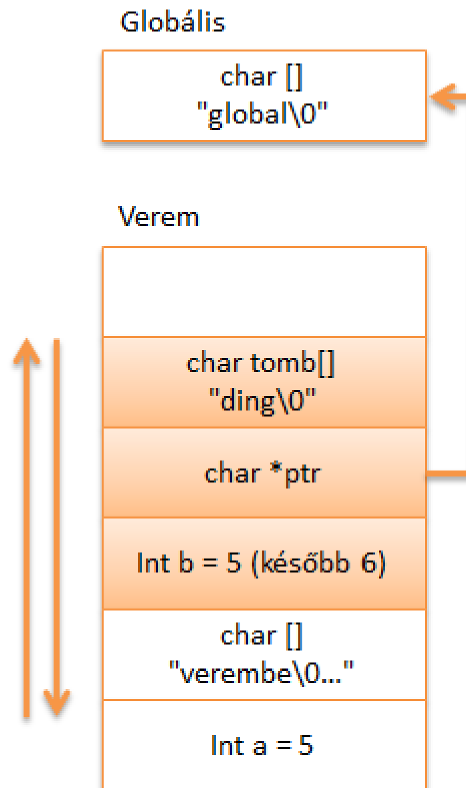
Függvényhívás esetén a meghívott függvény bemeneteinek (paramétereinek) értékei, és a cím amire vissza kell térnie, egy új verem-keretbe (stack frame) másolódnak, illetve itt kerülnek létrehozásra a meghívott függvény lokális változói is.

Ha egy függvény belsejének végrehajtásába kezdünk, akkor a verem tetején létrejönnek a függvény lokális változói, ha pedig a függvényből visszatérünk, akkor ezek a változók megszűnnek. Az adott függvényhíváshoz tartozó memóriaterület a veremben a stack frame. A veremben minden függvény számára csak a saját lokális változói látszódnak. Ha a függvény saját magát hívja meg, akkor különböző - és egymástól független - példányok keletkeznek a lokális változóiból.

Operációs rendszerek esetén minden végrehajtási szál kap egy saját vermet. Beágyazott rendszerek illetve mikrokontrollerek esetén a verem a megszakításkezelésben lát el fontos szerepet. A verem memóriakezelés általában gyorsabb, mint a halom.

A verem működését a következő C nyelven írt mintakód szemlélteti (3.2. ábra):

```
// Teszfüggvény
void fv(int b)
{
      char *ptr = "global";
      char tomb[] = "ding";
      b = 6;
}
// Az alkalmazás belépési pontja
int main()
{
      int a = 5;
      char s[50] = "verembe";
      fv(a); // Átadjuk az „a” értékét a fv() függvénynek
      return 0;
}
```



3.2. ábra A veremhez tartozó mintaprogram memória ábrája

Ennél a példánál a következő módon alakul a memóriaterületek tartalma: a program indításkor a „main” függvényt kezdi el végrehajtani. A main függvénynek két lokális változója van, egy egész („a”), és egy ötvenelemű karakter tömb („s”). A karakter tömb maga, vagyis az egyes karakterek is a veremben helyezkednek el.

Ha meghívódik az „fv” függvény, akkor induláskor létrejönnek a veremben a bemenetei és a lokális változói, a „b” nevű egész, a „ptr” nevű mutató és a „tomb” nevű tömb. A „ptr”-rel megint csak egy mutató kerül deklarálásra, nem pedig egy tömb! A verembe, amely a lokális változókat tárolja, így csak a mutató kerül, amely be is állítódik a „global” szót tartalmazó tömbre.

A tömb az előzőekhez hasonlóan a globális memóriaterületre került, és a program egész futása alatt létezik. A „char tomb[] = "ding"” sorral azonban nem mutató, hanem egy tömb kerül deklarálásra, így annak tartalma is a verembe kerül, még úgy is, hogy nem lett hozzá méret megadva, hanem azt az inicializáló karakterlánc alapján számolja a fordító.

A „tomb” a „ptr” és „b” változók csak addig léteznek, amíg a függvény belsejében vagyunk. Ha visszatér a függvény, és újra a „main”-be kerül a végrehajtás, akkor már nem fognak létezni. A „b = 6” emiatt értelemszerűen nem a „main”-ben deklarált „a” változót módosítja, hanem a veremben lévő „b” másolatot.

3.1.1.3 Halom alapú memóriakezelés

A dinamikus memóriaterület, vagyis a halom (heap) olyan terület, amelyből egy adott nagyságú részt a program futása közben le lehet foglalni, és ha már nem kell, akkor fel lehet szabadítani. Így lehetőség van akkora memóriaterület lefoglalására, melynek nagysága a

program írása, illetve fordítása közben még nem ismert. Az adott terület lefoglalásakor a lefoglalásért felelős függvény többnyire mutatót (pointert) ad vissza arra a memóriacímre, ahol a kontroller megfelelő nagyságú területet talált. Amikor a lefoglalt memória területre már nincs szükség, akkor az felszabadítható, pontosabban fel kell szabadítani.

A halomhoz mindig hozzátartozik a szabad és használatban lévő memóriablokkok/területek listája, melyre azért van szükség, hogy a dinamikus memóriafoglalás során mindenképpen egy olyan memóriaterület kerüljön lefoglalásra, amely még nincs használatban. Az esetek többségében a blokklista maga is a halomban van tárolva, gyakran a többi blokk között. Mivel a foglalás előtt meg kell keresni a megfelelő blokkot illetve területeket, lassabb szokott lenni, mint a verem.

A halom kezelése C-ben a malloc() és free() függvények hívásával, C++ban pedig a new és a delete operátorokkal történik. A foglalás során a foglaló (malloc(), new) keres egy megfelelő nagyságú szabad blokkot, majd frissíti a blokkok listáját, jelezve az újonnan lefoglalt blokk helyét. C-ben a halomból az adatokat mindenképpen kézzel kell törölni, erre szolgál a free() függvény.

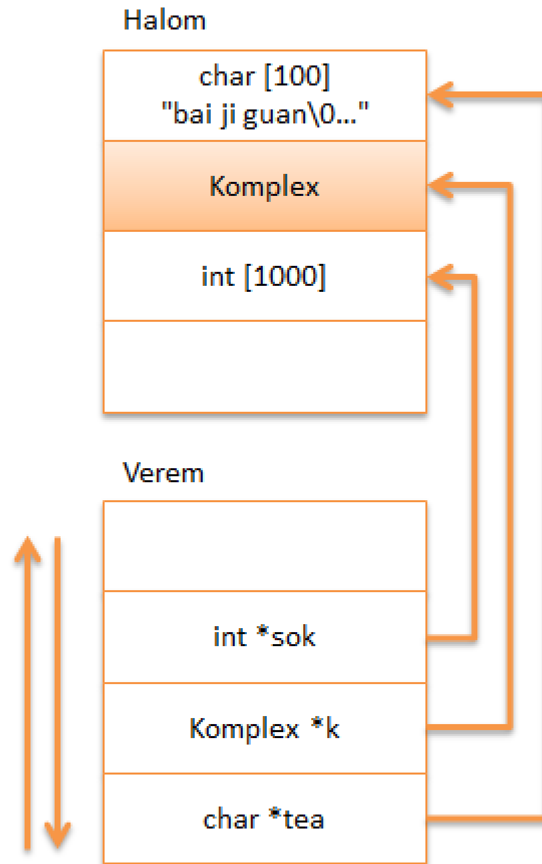
A következőkben egy-egy példa fogja szemléltetni a halom alapú memóriakezelést C, valamint C++ programozási nyelven (3.3. ábra).

```
// C

int main()
{
char *tea = malloc(100 * sizeof(char));
Komplex *k = malloc(sizeof(Komplex));
int *sok = malloc(1000 * sizeof(int));
strcpy(tea, "bai ji guan");
free(k);
free(tea);
free(sok);
return 0;
}
```

```
// C++

int main()
{
char *tea = new char[100];
Komplex *k = new Komplex;
int *sok = new int[1000];
strcpy(tea, "bai ji guan");
delete k;
delete[] tea;
delete[] sok;
return 0;
}
```



3.3. ábra A halomhoz tartozó első mintaprogram memóriaábrája

A fenti C illetve C++ nyelven írt példa kódokban először is deklarálva lett egy „char *” típusú mutató „tea” néven. Ilyenkor maga a mutató a veremben jön létre, de egy dinamikusan lefoglalt memóriaterületre mutat, amely száz karaktert képes tárolni. A dinamikus memóriaterület foglalásra C nyelven legtöbbször a malloc() függvény, C++ nyelven pedig a new[] operátor alkalmazható. A dinamikus memóriafoglalásnak köszönhetően a száz karakternyi hely a dinamikus memóriaterületen, a halomban foglalódik le.

A mintakód következő sorában egy Komplex számra mutató pointer kerül deklarálásra valamint ebben a sorban megtörténik a halomban történő helyfoglalás egy darab Komplex objektumnak. Ezt követő sor megint csak egy mutatót deklarál, valamint helyet foglal neki a veremben. Mint látható a mutatónak akkor lesz értelme, ha valami hasznos helyre mutat. Jelen esetben egy nagy, 1000 egészet tartalmazó, dinamikusan foglalt tömbre.

A lefoglalások után következő első hasznos sorban a lefoglalt „tea” karaktertömbbe lesz átmásolva egy karaktersor. Jelen esetben a másolandó karaktersor hossza 11 betű, illetve tartalmaz még egy lezáró nullát, vagyis 12 karakterből áll. Mivel a korábbiakban 100 karakter került lefoglalásra, így elegendő hely van a másolásra. Maga az eredeti karaktersor egyébként az előző példákhoz hasonlóan a globális memóriaterületen helyezkedik el, névtelenül. Onnan másolódik át jelen esetben a halomban lefoglalt területre.

Ezután következnek a lefoglalt területek felszabadításai. Először a Komplex típusú adatnak lefoglalt „k” memóriaterület kerül felszabadításra. Ezt érdemes egyből megtenni, amikor már

nincsen szükség az adott változóra. A „k” mutató ezután továbbra is oda mutat, ahol az a komplex szám volt, de ezután már nem szabad hivatkozni a területre, hiszen azt a felszabadítás után már más adatok tárolására újra hasznosíthatóvá vált. Ezt a szabályt mindig, minden körülmények között be kell tartani, hiszen nem lehet tudni, hogy a felszabadított memóriacím alatt milyen adat tárolódik. Lehet, hogy az adott program azóta még nem foglalt le további memória területet, ugyanakkor a program lehet többszálú, és akkor egy másik szálban bármikor lefoglalódhat és felülíródhat a már felszabadított memóriaterület, sőt akár az operációs rendszerhez is visszakerülhetett, és egy másik program használja, így nagy eséllyel már nem a felszabadítás előtti adatokat tartalmazza.

Sokszor szokták a felszabadított mutató értékét 0 értékre állítani ezzel jelezve, hogy nem mutat érvényes adatterületre, ugyanakkor mikrokontrollerek esetén illetve több operációs rendszernél is a nullás memóriacím is érvényes és elérhető cím.

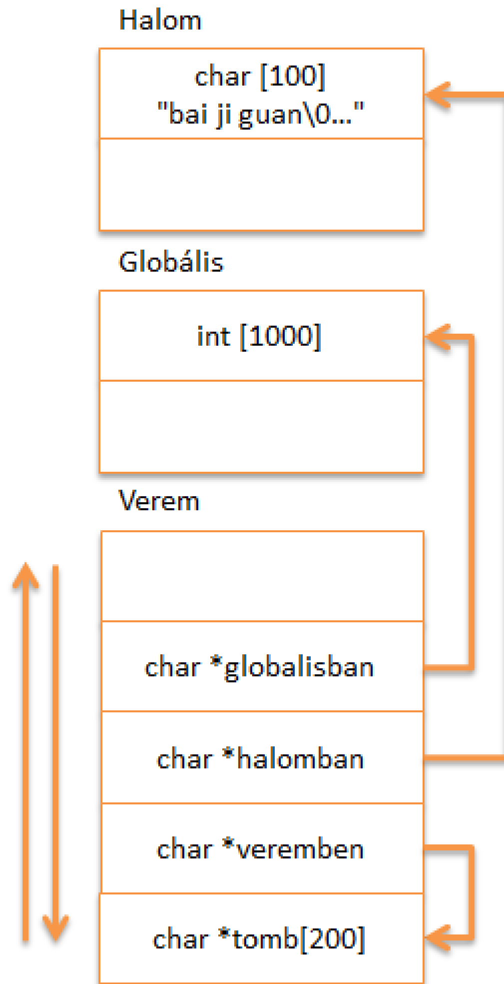
A többi terület felszabadítása hasonló módon történik. C++ nyelven, tehát amikor csak egyetlen Komplexnek kell helyet foglalni, akkor a new, felszabadításhoz pedig a delete operátort szokás használni.

Ha tömböt kell lefoglalni, akkor foglalásakor a new[], felszabadításkor pedig a delete[] operátort szokás alkalmazni. A szimpla és a tömb típusú adatokhoz tartozó operátorokat nem szabad keverni, amely memória terület a new operátorral lett lefoglalva, azt később delete operátorral, ami pedig new[] operátorral lett lefoglalva, azt pedig később delete[] operátorral kell felszabadítani. Ez lényeges mivel a new char és a new char[1] kifejezések nem ugyanazt jelentik. Ugyanakkor erre a programozónak figyelni kell, mivel a mutatón nem látszik, hogy az egyetlenegy adatra mutat, vagy egy tömbre. Vagyis egy önálló Komplex objektum memóriacíme, és egy Komplex tömb memóriacíme ugyanaz a típus, vagyis Komplex*. Ugyanígy egy mutatón nem látszik az, hogy dinamikusan foglalt memória területre vagy egy, a globális memóriaterületen, esetleg a veremben elhelyezkedő változóra mutat.

Fontos továbbá, hogy csak azt a memóriaterületet kell kézzel felszabadítani, aminek a foglalása dinamikusan történt, mivel a többi felszabadításáról a fordító gondoskodik. A veremből akkor törlődnek az adott változók, amikor vége a függvény végrehajtásának, valamint a globális memóriaterületről akkor törlődnek a változók, amikor a program végrehajtása befejeződik.

A második példa a korábban említett globális és a dinamikusan lefoglalt területre mutató pointer közötti különbségekre mutat rá (3.4. ábra).

```
char global[1000];
int main(){
     char tomb[200];
     char *veremben;
     char *heapen;
     char *globalisban;
     halomban = malloc(100);
     globalisban = global;
     veremben = tomb;
     free(heapen);
     return 0;
}
```



3.4. ábra A halomhoz tartozó második mintaprogram memóriaábrája

A „global" tömb a globális memóriaterületre kerül elhelyezésre, erre mutat a „globalisban" mutató. A „tomb" tömb a veremben helyezkedik el, a „veremben" mutató pedig a „tomb" változó első elemének a címére mutat. A „halomban" mutató a veremben tárolódik, és a malloc() függvény segítségével a halomban lefoglalt területre mutat.

A C++ szabvány megkülönbözteti a C stílusú malloc – free alapú memóriafoglalást a C++-os new – delete objektum orientált operátorokétól. A malloc – free páros által használt memóriaterületet halomnak (heap), míg a new – delete operátorok által használtat pedig szabad tárnak (free store) nevezi. Ezek persze lehetnek közösek, és egy programban lehet használni egyszerre mind a kettőt. Viszont, ami a malloc() függvény segítségével lett lefoglalva, azt a free() függvénnyel kell felszabadítani, nem a delete operátorral. Ugyanez igaz fordítva is.

Más „magasabb szintű" programnyelv, mint például a Java vagy C# esetén is van dinamikus memóriakezelés, ugyanakkor ott a felszabadítást nem kell kézzel elvégezni, hanem egy úgynevezett szemétgyűjtő eljárás (garbage collector) végzi el automatikusan a felszabadítást. Ennek lényege, hogy a szemétgyűjtő meghatározza, hogy mely objektumok nincsenek használatban, majd felszabadítja az általuk lefoglalt memóriát. Összességében elmondható, hogy a szemétgyűjtési eljárás kiküszöböl pár tipikus programozási hibát, mint például a már

felszabadított memória területekre történő hivatkozást, a már nem használt memóriaterület fel nem szabadítását (úgynevezett "memóriaszivárgás"), vagy a memóriaterületek többszörös felszabadítását. Ugyanakkor elő is idéz néhány, első sorban teljesítménybeli problémát. Ennek oka, hogy a szemétgyűjtés nem determinisztikus, azaz nem lehet tudni mikor fog törlődni egy objektum, így nagyobb memóriaterületek is lefoglalva maradhatnak még egy ideig úgy, hogy valójában nincsenek használva. Valamint az is erőforrást igényel, hogy minden egyes objektumról meghatározásra kerüljön, hogy mikor nincsen már rá szükség.

Mind a kézi, mind az automatikus memória felszabadításnak megvannak az előnyei és a hátrányai. Talán elmondható, hogy a manuális felszabadítás többnyire hatékonyabb memória-felhasználást szokott eredményezni, ugyanakkor az automatikus többnyire biztonságosabb.

A grafikus programozási nyelvek, mint például a LabVIEW és Matlab/Simulink esetén általában nem kell a kézi memória-felszabadítással foglalkozni.

3.1.2 Háttértárak kezelése

A mikrokontrollereknél van beépített, nem felejtő memória, ugyanakkor a nagyobb beágyazott rendszereknél alkalmazott processzorok esetén nem áll rendelkezésre ilyen könnyen írható, vagyis programozható memória. Ezeknek a rendszereknek szüksége van külső adattárolóra a programok, illetve adatok tárolásához; valamint gyakran a mikrokontrollerekhez is szoktak kiegészítő háttértárolókat illeszteni nagy mennyiségű adatmentés céljából. Ezek lehetnek mind mágneslemezes (pl.: hagyományos merevlemezek), esetleg szalagos adattárolók (elsősorban nagy mennyiségű adat biztonsági mentésére szokták még manapság is használni őket), optikai lemezek (pl.: DVD), valamint Flash alapú meghajtók is (pl.: SSD, SD-kártya).

3.1.2.1 Háttértár-kiosztási stratégiák

A háttértár-kiosztási stratégiák az állományoknak a fizikai adathordozón történő elhelyezési módját határozza meg. Többfajta stratégia is ismert erre vonatkozóan:

- A folytonos kiosztás esetén minden állomány egymás után álló blokkok sorozatát foglalja el a lemezen. A katalógusba a blokk kezdőcímét és az elfoglalt blokkok számát kell felvenni. Ezen módszer előnye, hogy a szekvenciális és véletlen elérésű állományokhoz is kiválóan alkalmas, ugyanakkor sokszor bonyolult algoritmusok szükségesek a megfelelő méretű szabad területek megkeresésére. Emellett külső elaprózódás is felléphet: a szabad terület egy idő után sok használhatatlan kis részre esik szét. Ekkor ugyan még lehet, hogy elegendő lenne a teljes egészében rendelkezésre álló szabad hely az adott állomány eltárolásához, de az összefüggően rendelkezésre álló hely ehhez már nem elegendő. Ezt a problémát tömörítéssel illetve töredezettség-mentesítéssel valamelyest lehet orvosolni, ugyanakkor ez lassú és időigényes folyamat is lehet. További hátránya, hogy problémás az állományok bővítése, mert gyakran át kell másolni egy nagyobb üres helyre, ha az adott állomány mögött rendelkezésre álló tárhely már nem elegendő a bővítéshez. Nagyobb helyet lehet lefoglalni, mint ami szükséges, de egyrészt nem biztos, hogy később tényleg bővítve lesz az állomány, ekkor ez felesleges helypazarlás, másrészt nincs arra garancia, hogy a jövőbeli bővítéskor a lefoglalt nagyobb hely tényleg elegendő lesz.

- A láncolt kiosztásnál minden egyes állomány blokkok láncolt listája, azaz minden blokk végén van egy mutató, mely a következő blokkra mutat. A katalógus az első és az utolsó blokk címét tartalmazza. Előnye, hogy a szabad helyek aprózódásának problémája nem jelentkezik, illetve nem gond az állományok bővítése. Hátrány, hogy csak szekvenciális állományokra jó, valamint sebezhető, vagyis ha egy mutató elveszik, elvész az állomány egész hátralévő része. Hiszen ahhoz, hogy egy adott blokk megtalálható legyen, az első blokktól kiindulva végig kell tudni menni a keresett blokkig a mutatók segítségével.
- Az indexelt kiosztás gyakorlatilag a láncolt kiosztás módosított változata. A fő különbség a két mód között, hogy ennél a változatnál a mutatók egy külön indextáblában kerülnek eltárolásra oly módon, hogy az indexblokk "i"-edik eleme az állomány "i"-edik blokkjára mutat. Előnye, hogy nem csak szekvenciális állományokra jó, így ez a legrugalmasabb megoldás. Hátránya az indexblokkokból ered, ugyanis nem biztos, hogy az indexblokk belefér egy blokkba (hosszú állományok esetén). Ilyenkor a megoldás az indexblokkok láncolt listája lehet. Ha a fizikai blokkméret sokkal nagyobb, mint az indexek elhelyezéshez szükséges terület, akkor belső elaprózódás lép fel kisméretű állományoknál.

3.1.2.2 Katalógusszerkezetek (könyvtárszerkezetek)

A katalógusszerkezeteknek, illetve könyvtárszerkezeteknek az állományok rendezésében van komoly szerepük. Összetettebb, több felhasználós operációs rendszerek esetén bonyolult módon is történhet a könyvtárszerkezetek kialakítása, illetve különböző operációs rendszerek több eltérő stratégiát is alkalmazhatnak:

- A kétszintű katalógus esetén minden felhasználónak saját katalógusa van (ezek neveit tartalmazza a felső szintű katalógus). E saját katalógusokban helyezhetik el a felhasználók saját állományaikat. Ezen kívül van egy "rendszerkatalógus", ahol a közös használatú állományok találhatók meg. Egyszerű, de merev rendszer.
- A fa struktúrájú katalógus a kétszintű módszer továbbfejlesztése. A felső szintű (gyökér) katalógusban állományok mellett újabb katalógusok találhatók, majd ezekben a katalógusokban szintén állományok és újabb katalógusok vannak és így tovább. Elég rugalmas, gyakran használt megoldás. Ez a legtöbb gyakorlatban használt fájlrendszer alapelve.
- A ciklusmentes gráf a fa struktúrájú katalógus továbbfejlesztése. Ha egy állomány több katalógusban is szerepel, a több másolatnak az egyes katalógusokban külön-külön való tárolása helyett az adott állományból csak egy példányt tárol, és a megfelelő katalógusokban speciális bejegyzésekkel, az ún. hivatkozásokkal (linkekkel) mutat e közös példányra.

3.1.2.3 Lemezes háttértár ütemezési stratégiák

A lemezes háttértáraknál nagyon lényeges, hogy a beérkező olvasási illetve írási kérések hogyan kerülnek kiszolgálásra. Mivel itt állandóan forgó lemez(ek) felett mozog(nak) író illetve olvasó fej(ek), ezért lényeges, hogy mikor melyik helyre fog elmozdulni az író illetve olvasó fej. Ez kulcsfontosságú, mivel a fej mozgatásának ideje nagyságrendekkel több időt

vehet igénybe, mint az adatok írása illetve kiolvasása, így egy megfelelő stratégiával sokat lehet gyorsítani az állományok elérésén. Az ütemezésnek abban az esetben van igazán fontos szerepe, ha egyszerre több fájlművelet is történik, azaz egyszerre több írási és olvasási kérelem is érkezik. A főbb stratégiák a következők szoktak lenni:

- Az FCFS (First-Come, First-Served) esetén amelyik kérés előbb jött, az lesz előbb kiszolgálva. A kérelmek egy FIFO (First In, First Out) sorba, (azaz ami legkorábban került be, az kerül ki legelőször) kerülnek be, és mindig a FIFO elejéről vesszük ki a következő kiszolgálandót. A legrosszabb hatásfokú stratégia a fejmozgás szempontjából.
- Az SSTF (Shortest Seek Time First) esetén a legkisebb fejmozgást igénylő kérést részesíti előnyben, azaz mindig azt az igényt elégíti ki, amelyhez a fej éppen a legközelebb van. Ritkán szokták használni a gyakorlatban, inkább csak speciális esetekben alkalmazzák, mivel nagy a kiéheztetés veszélye, vagyis a fej jelenlegi állásától messze levő kérésig lehet, hogy soha nem jut el.
- A SCAN egy pásztázó módszer. A fej a két végállása között folyamatosan ingázik és kielégíti az éppen aktuális pályára vonatkozó azon igényeket, amelyek a pásztázás kezdetekor már fennálltak. Ez azt is magába foglalja, hogy a kiszolgálás közben érkező új igényeket csak a következő "körben" szolgálja ki, így kerüli el a kiéheztetést. Hátránya, hogy a lemez szélein lévő állományokhoz tartozó kéréseket ritkábban szolgálja ki.
- Az N-SCAN (N lépéses pásztázó) szintén egy pásztázó eljárás. A hagyományos pásztázó eljárástól abban tér el, hogy egy irányba mozogva csak maximum "N" darab igényt elégít ki minden pályán, ennek következtében az átlagos várakozási idő körülbelül ugyanaz, mint a hagyományos SCAN esetében, de a szórása kisebb.
- C(ircular)-SCAN (egyirányú pásztázó) az előző kettőhöz hasonlóan szintén egy pásztázó módszer. Ennél az eljárásnál a kérések kiszolgálása mindig csak az egyik irányú fejmozgásnál történik, valamint elkerüli még a szélső pályák háttérbe szorítását is.

3.1.2.4 Virtuális tárkezelés

A virtuális tárkezelés a memóriakezelést egészíti ki a háttértárak megléte által nyújtott lehetőségek kihasználásával. Értelemszerűen, amikor az adott rendszer nem rendelkezik háttértárral, akkor a virtuális kezelés hagyományos formája nem valósítható meg, továbbá operációs rendszerenként módosulhatnak a virtuális tárkezelési stratégiák, illetve módszerek.

Magának a virtuális tárkezelésnek az alapelve az, hogy ha túl kevés hagyományos memória áll rendelkezésre, akkor a háttértárat is lehet memóriaként kezelni. Ugyan a háttértárak jóval lassabb írási és olvasási időt tesznek lehetővé, de így legalább a rendszer memóriakorlátjai kitolhatóak. Ez úgy valósítható meg, hogy a virtuális tárkezelés esetén a felhasználó által kiadott memóriacímek egy háttértáron (virtuális tárban) levő címeknek felelnek meg, és e címtartománynak csak bizonyos részei találhatóak meg a műveleti memóriában.

A virtuális tárkezelés alkalmazásának számos előnye van, többek között a multiprogramozott rendszerek esetén. Ha a folyamatoknak csak egy-egy részét tároljuk a műveleti memóriában, akkor több folyamatot futtathatunk párhuzamosan, illetve ma már a processzorok

címtartománya is kellően nagyméretű ahhoz, hogy ekkora operatív memóriát ne lehessen, vagy ne legyen szükséges kiépíteni. A virtuális tárkezeléssel elérhető, hogy viszonylag kisméretű operatív tároló használata esetén is a felhasználó úgy lássa, mintha egy teljes címtartomány méretű operatív tárat használna.

A virtuális tárkezelés leggyakrabban használt formája a lapszervezésű virtuális tár. Ilyenkor a virtuális tárat és az operatív tárat is fel kell osztani egyforma méretű egységekre, azaz lapokra, és az operatív memóriában egy laptáblát kell létrehozni ezen lapokhoz.

Ez a laptábla tartalmazza azt, hogy az adott lap az operatív tárban található-e vagy sem, illetve, hogy ha megtalálható, akkor mi a lap kezdőcíme az operatív tárban. (Valamint tartalmaz egyéb vezérlő biteket is.). A processzor által kiadott logikai (virtuális) címet logikailag két részre lehet szétválasztani. A felső rész kiválaszt egy bejegyzést a laptáblából. Itt megtalálható a lap operatív tárbeli kezdőcíme, ehhez hozzáadva a cím második felét, az úgynevezett lapon belüli eltolást, így megkapható a keresett memóriahely címe az operatív tárolóban. A címszámítás gyorsítására az utoljára használt néhány lap címét tartalmazó asszociatív tárat szoktak használni.

Virtuális tárkezelés használata esetén előfordulhat, hogy a processzor által kiadott címet tartalmazó lap nem található az operatív memóriában, ez a laphiba. Ezt kezelni kell, melyre leggyakrabban az alábbi elvi eljárást szokás alkalmazni:

- Ellenőrizni kell, hogy a kiadott címet az adott folyamat használhatja-e.
- A kérdéses lapot be kell olvasni a műveleti memóriába (természetesen ez azzal járhat, hogy előtte egy bent lévő lapot ki kell emelni a háttértárba), majd módosítani kell laptáblát.
- Meg kell ismételni annak az utasításnak a végrehajtását, amelynél a laphiba fellépett.

A virtuális tárkezelés esetén többféle lapozási stratégiát is lehet alkalmazni, azaz, hogy laphiba esetén mely módon legyen a probléma lekezelve:

- A hagyományos FIFO (First In, First Out, azaz ami legkorábban került be, az kerül ki legelőször) esetén a behozott lapok számai egy FIFO tárban kerülnek letárolásra. Laphiba esetén a FIFO sor elején álló (azaz a legrégebben behozott) lap lesz kiemelve, és az újonnan behozott lap sorszáma kerül a FIFO sor végére. Előnye, hogy nagyon egyszerű, ugyanakkor önmagában ritkán szokták alkalmazni, mivel sok laphibát generál. Hiszen elképzelhető, hogy egy lap már régóta be lett hozva, de még mindig használatban van. Ez tipikusan igaz a magát az operációs rendszert „tartalmazó” lapokra, amelyeket ez az algoritmus így állandóan „kilapozna”.
- Az OPT (optimális) módszer esetén az új lap mindig annak a lapnak a helyére kerül be, amelyre a legkésőbb lesz (újra)hivatkozás. Előny, hogy ez adja a minimális laphiba számot, ugyanakkor a gyakorlatban megvalósíthatatlan, mivel nem lehet előre tudni a lapokra való hivatkozások sorrendjét. Ezért csak az egyéb stratégiák jóságának vizsgálatához, referenciaként használják.
- Az LRU (legrégebben használt) eljárás esetén az új lap mindig annak a helyére kerül beemelésre, amelyre a legrégebben történt hivatkozás, azaz amely a legrégebben volt használva. Az algoritmus alapja a lokalitási elv megfordítása, azaz ha egy lap már

régóta nem volt használva, akkor nagy valószínűséggel nem lesz rá szükség a későbbiekben sem. Előnye, hogy viszonylag jól közelíti az optimális módszert, ugyanakkor kiegészítő hardvert igényel, különben nagyon lassú lesz.

- A „Second chance”, vagy más néven második esély algoritmus esetén minden lapon van egy hivatkozás bit is, amelyet 1-re kell állítani minden alkalommal, amikor a lapra hivatkozás történik. Ez igény szerinti lapozásnál azt jelenti, hogy amikor beemelésre kerül az adott lap, akkor ezt a bitet rögtön 1-re kell állítani, hiszen azért kerül beemelésre a lap, mert hivatkoztak rá. Lapcsere esetén azt a lapot kell kicserélni, amelyik a legrégebb óta van bent (FIFO), de ha a FIFO módszerrel meghatározott lap hivatkozás bitje 1, akkor a lapot 0 értékű hivatkozás bittel át kell helyezni a FIFO végére, azaz csak 0 hivatkozás bitű lapot lehet lecserélni.

Virtuális tárkezelés esetén bekövetkezhet a „vergődés” jelensége. A multiprogramozott számítógéprendszerek esetén ez azt jelenti, hogy egy folyamat több időt tölt lapozással, mint saját hasznos tevékenységével. Ennek az az oka, hogy a folyamat kevesebb lapkerettel rendelkezik a „szükségesnél” az operatív memóriában. A „vergődés” a lapozási stratégia javításával többnyire megelőzhető.

3.1.2.5 A mágneses merevlemezes háttértárak szervezési elve

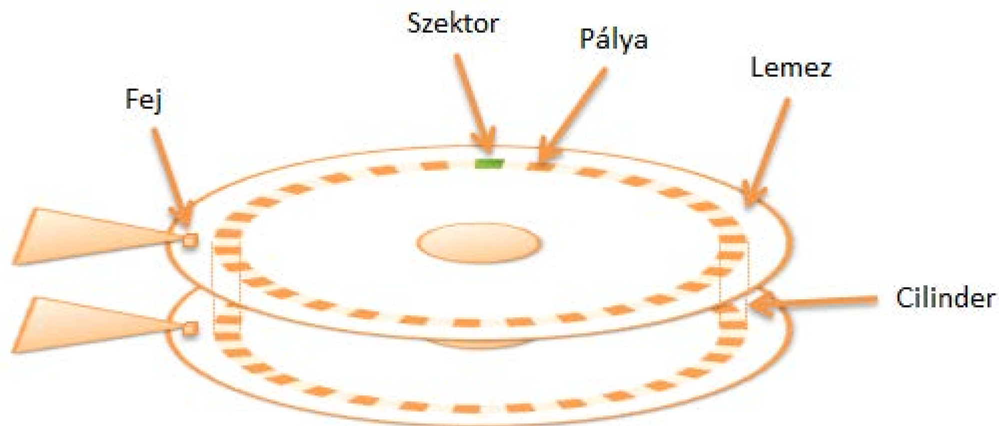
A mágneses merevlemezes háttértárak közé tartoznak a „hagyományos” háttértárolók. Széles körben elterjedtek a nagyobb méretű beágyazott rendszerek esetében. Ezen eszközökben egy vagy több lemez található, többnyire lemezenként egy olvasófejjel (3.5. ábra). A lemezeket koncentrikus körökre, sávokra vagy pályákra (track) lehet felosztani. Azon merevlemezes egységeknél, ahol több lemez helyezkedik el egymás alatt, az egymás alatti sávok összességét nevezzük cilindernek. Egyes esetekben lehetséges, hogy a lemez mindkét oldalát lehet íni illetve olvasni, ilyenkor is lehet cilinderről beszélni. A sávok további egységekre, az úgynevezett szektorokra vannak osztva. A 0. szektor helyzetét egy referencia-, vagy más néven indexlyuk jelzi. Az egyes szektorok között üres helyek („gap”-ek) találhatóak.

Minden pálya illetve szektor elején egy azonosító bitsorozat áll, amelyet a formázás során a formázó program ír fel. Az adatblokkok az egyes szektorokban helyezkednek el. Az adatblokkok méretétől függően lehetséges, hogy egy adatblokk:

- egyenlő egy szektorral,
- kisebb, mint egy szektor,
- nagyobb, mint egy szektor.

Az, hogy az egyes állományokhoz tartozó adatterületek hogyan helyezkednek el a lemezen, nagymértékben függ az alkalmazott háttértér kiosztási stratégiájától.

Jelenleg a mágneses adattárolási elv szerint működő merevlemezek a legelterjedtebb háttértárak, bár a szilárdtest meghajtók (SSD) egyre jobbá és olcsóbbá válnak.

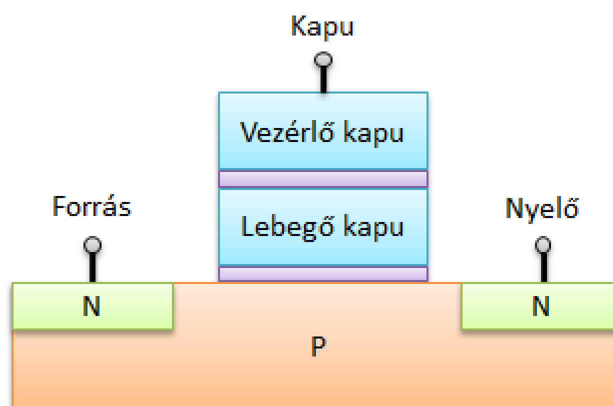


3.5. ábra Hagyományos mágneses háttértár felépítése

3.1.2.6 Flash memóriák

A Flash memória egy nem felejtő, elektronikusan írható és törölhető memória, az EEPROM memóriák továbbfejlesztéseként jött létre. Két típusát különböztetjük meg: NOR (Not OR) és NAND (Not AND) flash memóriákat. Mindkét memória típus lebegőkapus tranzisztorokból épül fel, a főkülönbség a kettő között a tranzisztorok elrendezésében van.

A lebegőkapus tranzisztor lényegében egy MOSFET, amely két, egymástól szigetelő anyaggal elválasztott Gate (kapu) elektródával rendelkezik (3.6. ábra). A tárolás működése a következő: ha a drain (nyelő) és a gate (kapu) elektródára pozitív feszültséget, a source (forrás) elektródára pedig földpotenciált kapcsolnak, akkor elektronok kezdenek el áramlani a csatornában. Ha a kapu elektródára kapcsolt pozitív potenciál elegendően nagy, akkor hatására az elektronok képesek lesznek átjutni a csatornát és a lebegőkaput elválasztó szigetelő rétegen. A feszültség lekapcsolása után azonban az elektronok csapdába esnek, és nem lesznek képesek visszafele átjutni a szigetelésen. Ezáltal a lebegőkapus tranzisztor képes lesz egy adott állapotot eltárolni. Az elmondottak visszafele is érvényesek, azaz a lebegőkapuban rekedt elektronokat ki tudjuk lökni egy negatív feszültség segítségével.



3.6. ábra Lebegő kapus MOSFET, a Flash memóriák alapegysége

A NOR és a NAND Flash memóriák a nevüket onnan kapták, hogy a tranzisztorok úgy vannak elrendezve a szilícium lapkán, hogy az adott névnek megfelelő logikai funkciót látják el. Mindkét típusnak megvan a maga előnye és hátránya egyaránt. A NOR típusú memóriánál

lehetőség van bájtonként elérni a memóriában tárolt információt és igen kis válaszidővel rendelkezik. Hátránya, hogy adott méretű szilícium lapkán kevesebb tároló cella alakítható ki, illetve előállítása drágább a NAND memóriánál. NOR Flash memóriát ott érdemes használni, ahol szükséges a szabad elérés (random access, RAM-ként való alkalmazás) és a gyorsaság, illetve nincs szükség nagy mennyiségű adat olvasására. Ilyen alkalmazási lehetőség például beágyazott rendszerekben található mikrovezérlők belső Flash memóriája is.

A NAND típusú Flash memóriák nagy sebességgel, és akár 4-szer nagyobb tároló kapacitással rendelkeznek ugyanakkora szilícium lapkán, mint a NOR típusú memóriák. Ezt a típusú memóriát ott használják, ahol nincs szükség a bájtonkénti elérésre, illetve a folyamatos írás és olvasás kiszolgálására. Ugyanakkor nagy mennyiségű adat egyszerre történő mozgatása is szükséges. NAND típusú memória található a pendrive-okban, SD kártyákban.

4 Kommunikáció

Egy beágyazott rendszernek adatcserét kell folytatnia több más részegységgel illetve eszközzel. A különböző beágyazott rendszerek különböző kommunikációs interfészekkel tartják egymással vagy más rendszerekkel a kapcsolatot. Gyakran megkülönböztetik a rövid távolságon belül - például egy közös NYÁK-on (Nyomtatott ÁramKör) - elhelyezkedő kontrollerek közötti kommunikációról (ilyenre szokták tipikusan alkalmazni az SPI és az I^2C protokollokat), valamint nagyobb távolságon (több méter vagy még nagyobb távolságon) történő kommunikációt (ilyenre szokták tipikusan alkalmazni a CAN, FlexRay, MOST, stb. protokollokat).

A különböző kommunikációs protokollokat többféleképpen is szokták kategorizálni. Az egyik ilyen csoportosítás a kommunikáció irányára vonatkozó kategorizálás. Ezek szerint lehet beszélni a szimplex kommunikációról, melynél minden jel illetve jelzés, információ csak és kizárólag egy irányba áramlik. A kétirányú kommunikációnál lehet fél-duplex (half-duplex) valamint teljes-duplex (full-duplex) kommunikációról beszélni. Előbbinél egy időben csak egy irányba használható a kommunikációs csatorna, azaz a két átviteli irány azonos időben szimultán nem használható. Az utóbbinál megengedett az egy időben mindkét irányba történő kommunikáció.

Egy másik kategorizálási lehetőség, hogy a kommunikáció során van-e összehangoló órajel, vagy nincs. Az előbbi a szinkron (pl.: SPI, I^2C), míg az utóbbi az aszinkron (pl.: RS-232, CAN) kommunikáció. Aszinkron kommunikáció esetén nincs közös órajel, ami az adást és a vételt összehangolja. Ezért a kommunikáció csak előre meghatározott sebességeken történhet, ezért adó és vevő közötti „szinkronizációhoz” nagyon pontos időzítés kell, valamint a buszon levő összes eszköz esetén ugyanazokat a beállításokat kell alkalmazni.

4.1 Az autóiparban leggyakrabban alkalmazott kommunikációs protokollok

Az autóiparban számos különböző protokollt alkalmaznak az eltérő igények kielégítésére. Ezek közül talán a legismertebbek és legelterjedtebbek az SPI, I^2C, UART/RS-232, CAN, LIN, FlexRay, CANopen és a MOST.

4.1.1 UART és RS-232

A mikrokontrollerek között az egyik legelterjedtebb kommunikációs protokoll az UART (Universal Asynchronous Receiver Transmitter), mely soros, aszinkron átvitelt tesz lehetővé. Alapértelmezetten három vezeték szükséges a működéséhez: egy fogadó (RXD), egy küldő (TXD), valamint egy földelés (GND) vezeték. A vonalak feszültségszintjei a szabványos TTL (Tranzisztor - Tranzisztor Logika) jelszinteknek felelnek meg, azaz a logikai 0, vagyis az alacsony jelszint: 0 V – 0,8 V közötti tartománynak, míg a logikai 1, azaz a magas jelszint a 2,4 V – 5 V közötti feszültségtartománynak felel meg.

Az RS-232 (Recommended Standard 232) nagyon hasonlít az UART-hoz, ezért gyakran együtt tárgyalják őket. Ugyanakkor lényeges különbségek is vannak, elsősorban a feszültségszintekben. A hagyományok szerint a két kommunikációs felet DTE-nek (Data Terminal Equipment) valamint DCE-nek (Data Communication Equipment) szokták nevezni,

mely elsősorban a lábkiosztásnál lényeges. Az RS-232 esetén minimum három vezeték szükséges - akárcsak az UART esetében - , ugyanakkor definiál több kiegészítő vezetéket is, melyek elsősorban a kifejlesztés eredeti céljára utalnak, vagyis a modemek, terminálok összekapcsolására:

- DCD (Data Carrier Detected): A DCE kapcsolódik a külső kommunikációs (telefon) vonalhoz.
- RXD (Receive Data): A DCE felől a DTE felé adatok küldésére szolgáló vonal.
- TXD (Transmit Data): A DTE felől a DCE felé adatok küldésére szolgáló vonal.
- DTR (Data Terminal Ready): Azt jelzi, hogy a DTE jelen van-e.
- GND (Ground): Földelés.
- DSR (Data Set Ready): Azt jelzi, hogy a DCE készen áll az adatok illetve parancsok fogadására.
- RTS (Request To Send): Azt jelzi, hogy a DTE adatot szeretne küldeni a DCE felé.
- CTS (Clear To Send): Azt jelzi, hogy a DCE kész az adatok fogadására.
- RI (Ring Indicator): A DCE hívó jelet észlelt a bejövő kommunikációs (telefon) vonalon.

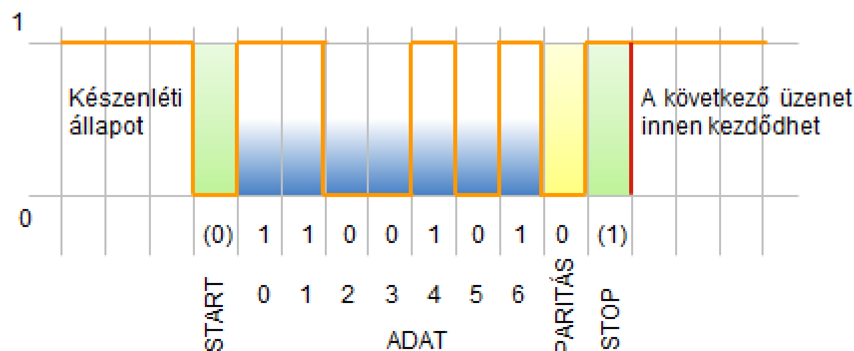
A fenti vonalak közül manapság inkább már csak a fogadó (RXD), egy küldő (TXD), valamint a földelés (GND) vezetékeket szokták használni, illetve még DTR, DSR, CTS, RTS abban az esetben, ha hardver alapú kézfogásra van szükség a kommunikáció során.

Az RS-232 esetén a vonalak jelszintjei jelentősen eltérnek az UART jelszintektől, mivel itt nem TTL jelszintek vannak, hanem a logikai 0, azaz alacsony jelszintnek a 3 V – 15 V közötti, míg a logikai 1, azaz magas szintnek a -3 V – -15 V feszültség tartomány felel meg.

Mivel az UART és az RS-232 is aszinkron protokoll, vagyis nincsen közös szinkronizáló órajel a felek között a kommunikáció alatt, így rendkívül fontos a kommunikációban résztvevő két fél megfelelő beállítása, máskülönben nem tudják megfelelően értelmezni az egymásnak küldött adatokat illetve vezérlő jeleket. Alapvetően a kommunikáció paraméterei a következőkkel jellemezhetőek:

- Adatátviteli sebesség, amely azt határozza meg, hogy a bitek milyen gyorsan követik egymást, azaz egy bit időegységben milyen „hosszú”. A szabványos értékek a következők szoktak lenni: 150, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200 bit/s.
- Adatbitek száma, azaz egy üzenetben hány darab adatbit található. A szabványos értékek a következők szoktak lenni: 5, 6, 7, 8, 9.
- Paritásbit, azaz páros, páratlan vagy semmilyen paritás bit alkalmazandó.
- Stop bitek száma, azaz milyen „hosszú” legyen egy stop bit. A szabványos értékek a következők szoktak lenni: 1 bit, 1,5 bit, 2 bit.

Az UART illetve az RS-232 üzenetformátumot is definiál, melyet komolyan befolyásolnak a kommunikációra vonatkozó fentebb leírt beállítási lehetőségek (4.1. ábra)



4.1. ábra UART/RS232 protokoll által definiált üzenetformátum

Amikor nincs adatküldés, az adatvezetéken logikai 1 szint van a vonalon. Ha az eszköz adatot akar küldeni, akkor azt egy logikai 0 szinten levő start bit segítségével jelzi. A start bitet folyamatosan, sorban egymás után követik az adatbitek. Az adatbitek végén pedig paritásbit is állhat. Az adatküldés befejeztével az átvitel végét egy stop bit jelzi, ami mindig logikai 1 szinten van. A stop bit után azonnal következhet egy újabb átvitel.

Az üzenet fogadás esetén a fogadó készülék várakozik az adatátvitel megkezdésére. Amikor lefutó élt detektál, akkor vár fél bit időtartamig és újra beolvassa az RXD vonalat, hogy meggyőződjön arról, hogy biztosan start bit, és nem zaj érkezett. Ezután minden bitidő közepén mintát vesz a vevő a jelvezeték logikai szintjéből, így olvasva le minden bit értékét a stop bitig bezárólag.

Az RS-232 esetén hardveres kézfogás is lehetséges, melynek két fajtája van. Ez egyik a kapcsolat létrehozásáért felelős (DTR/DSR), míg a másik az adatátvitelt vezérli. Természetesen külön-külön, valamint együtt is alkalmazható a két eljárás. Abban az esetben, ha mindkét kézfogás használandó, a következők szerint néz ki a kommunikáció:

- Az adó jelzi DTR (Data Terminal Ready) vonalon a vevő felé, hogy készen áll a forgalmazásra.
- Az adó jelzésére a vevő a DSR (Data Set Ready) vonalon visszaigazolja, hogy szintén készen áll a kommunikációra.
- Az adó az RTS (Request To Send) vonalon jelzi, hogy adatot kíván küldeni.
- A vevő a CTS (Clear To Send) vonalon visszaigazolja, hogy készen áll az adat fogadására, illetve a kommunikációra.
- Megkezdődik a kétirányú adatforgalmazás a RXD (Receive Data) illetve a TXD (Transmit Data) vonalakon
- A vevő a kommunikáció alatt visszavonhatja a CTS jelet abban az esetben, ha nem tud adatokat fogadni (pl. megtelt a puffere).
- A CTS jel ismételt kiadásával a forgalmazás újraindul.
- Az adó az RTS jel megvonásával szintén jelezheti, hogy szüneteltetni akarja a kommunikációt.

Az UART esetében fontos megjegyezni, hogy sokszor, mint gyűjtőfogalmat használják a soros és párhuzamos interfészek közötti átalakításra, azaz például a belső buszrendszer, valamint egy soros kimenet között.

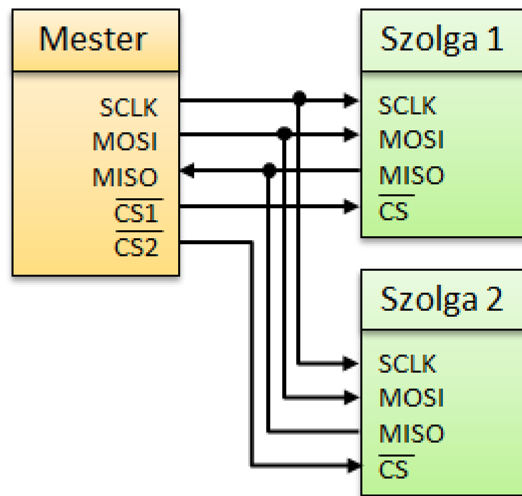
4.1.2 SPI

Az SPI (Serial Peripheral Interface Bus) egy szinkron, teljes-duplex, soros átviteli protokoll. Mester-szolga (Master-Slave) architektúrájú kommunikációt tesz lehetővé, azaz van egy kitüntetett eszköz a kommunikációban, amely vezérli a kommunikációt, azaz megadja, hogy melyik szolga eszköz kommunikálhat, valamint mivel szinkron kommunikációról van szó, ezért az órajelet is ez az eszköz szolgáltatja.

Az SPI-t esetenként szokták négy vonalas soros busznak (4 wire serial bus) is nevezni, ezzel megkülönböztetik a kettő, három, sőt az egy vonalon történő soros adatátviteltől. Nagy előnye ennek a kommunikációs protokollnak, hogy nagy sávszélességet biztosít: az órajel frekvenciájától függően akár 10 MBit/s is lehet. Viszonylag elterjedt, emiatt gyakran a legegyszerűbb kontrollerek is támogatják hardveresen. További jó tulajdonság, hogy az adatátvitelkor kiválasztható az átviteli formátum, ami azt jelenti, hogy nem csak 8 bites egységeket, azaz bájtokat, hanem akár 4 vagy 24 bites adategységeket is lehet küldeni. Hátránya viszont, hogy több vezetősávot igényel a nyomtatott áramköri panelokon, illetve relatíve kis hatótávolsággal működik összehasonlítva az RS-232-es vagy a CAN protokollokkal.

A négy vonalas soros busz elnevezés néha félrevezető is lehet, mivel az SPI-nak létezik 3 illetve 5 vezetékes változata is, azonban az alap változatához 4 vezeték szokott tartozni:

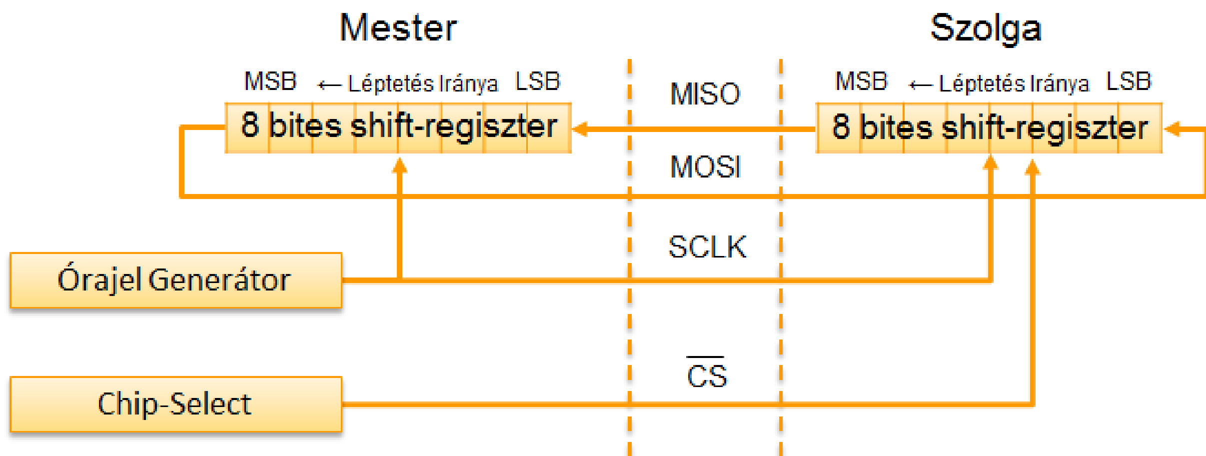
- Az SCLK vezetéken szolgáltatja a mester az órajelet, mely tipikusan 1-70 MHz szokott lenni.
- A MISO (master-in-slave-out) vagy más néven SOMI a mester felé menő kommunikációs vezeték, amelyen a szolga tud a mesternek adatot küldeni.
- A MOSI (master-out-slave-in) vagy más néven SIMO a szolga felé menő kommunikációs vezeték, amelyen a mester tud a szolgának adatot küldeni.
- A CS (chip select) vagy más néven SS (slave select) vezetéken tudja a mester engedélyezni az egyes szolga eszközöket, azaz ezzel tudja meghatározni kivel szeretne kommunikálni. A kiválasztó vonalak száma szab határt annak, hogy hány szolga eszköz csatlakozhat a hálózathoz, valamint ha csak egy szolga eszköz van rácsatlakoztatva a hálózatra, akkor nem kötelező a használata.



4.2. ábra Az SPI hálózat elrendezése két szolga esetén

Mivel mester-szolga jellegű, ezért a mester az aktív elem a rendszerben, vagyis ő kezdeményezi a kommunikációt, biztosítja a kommunikációs órajelet, valamint bekonfigurálja úgy a kommunikációt, hogy az a szolga eszköznek is megfelelő legyen. Ez elsősorban az órajel frekvenciát jelenti. Vagyis, hogy ne legyen túl magas frekvenciájú az órajel az adott szolga eszköz számára valamint, hogy le- vagy felfutó élre történjen az írás illetve olvasás.

Maga a kommunikáció „buffer-csere” jellegű, vagyis a mester és a szolga közötti adatcsere folyamán egyszerre, egy órajel ciklus alatt tolódik egy-egy adatbit a mestertől a szolga felé illetve a szolgától a mester felé (4.3. ábra). Tehát gyakorlatilag az ábrán látható két 8 bites shift regiszter úgy vehető, mint egy egyetlen 16 bitből álló körkörös shift regiszter, vagyis 8 órajel impulzus után az adat a mester és a szolga között kicserélődik



4.3. ábra Az SPI kommunikáció elve

A kommunikáció az alábbi lépések szerint történik:

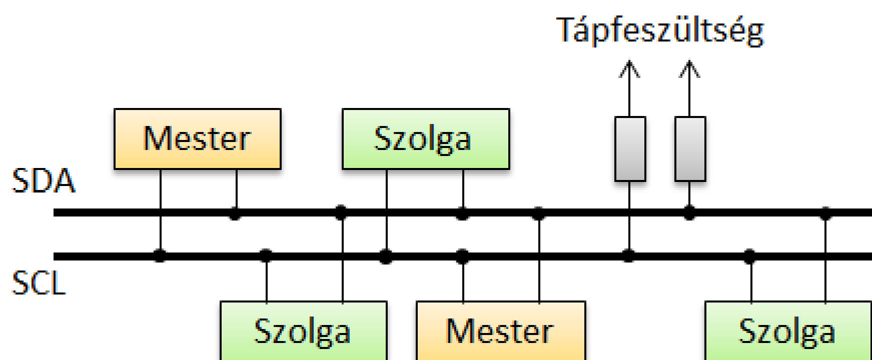
1. A mester a megfelelő szolga eszközhöz tartozó CS lábat aktívra állítja.
2. A mester és a szolga eszköz előkészíti a saját shift regiszterében a küldeni kívánt adatokat (az adat a mestertől a szolga felé mindig a MOSI vezetéken, a szolgától a mester felé pedig a MISO vezetéken áramlik)

3. A mester egy adatbitet ír a MOSI vezetékre, ezzel egy időben a szolga is egy adatbitet ír a MISO vezetékre.
4. Amikor a mester az SCLK vezetéken elküldi az adatcseréhez szükséges órajel impulzust, akkor beolvassa a MISO vezetéken lévő értéket (amit előzőleg a szolga írt rá), ugyanekkor a szolga is beolvassa a MOSI vezetéken lévő értéket (amit előzőleg a mester írt rá).
5. Adatbeolvasáskor a shift regiszter automatikusan továbblépteti a korábban beérkezett adatokat, helyet csinálva ezzel a bejövő adatnak (az SPI működési módtól függ, hogy adatbeolvasás az órajel impulzus felfutó vagy lefutó élére történik).
6. A 2-es ponttól ismételve a lépéseket az órajel minden egyes impulzusára az adatok bitenként elküldhetők.
7. Minden adatcsomag után a mester a CS vonalat magasra állítja, ezzel szinkronizálva a szolga eszközt.

Az átvitel során lehetőség van egyszerre több egységnyi adatot küldeni anélkül, hogy a szolga eszközzel meg kellene szakítanunk a kommunikációt. Ilyen esetben az órajel generálás tovább folyik, illetve a mester és a szolga küldheti a következő adatot.

4.1.3 I²C

Az I²C (Inter Integrated Circuit) egy soros, szinkron, fél-duplex kommunikációs protokoll. Ez a protokoll az SPI-hoz hasonlóan szintén megtalálható a legtöbb mikrokontrollerben. Noha több különböző gyártó által létrehozott protokoll is alapjául vette az I²C specifikációját, ezek azonban minimális átkonfigurálást igényelnek az I²C protokollhoz képest, és általánosságban elmondható, hogy kompatibilisek is egymással (SMBus, SCCB). Az I²C egy több mester és több szolga egységet is kezelni tudó protokoll, vagyis több mester és több szolga is lehet a hálózaton. A kommunikációhoz jellemzően két vezeték szükséges, egy szinkron órajel vezeték (SCL), valamint egy adatjel vezeték (SDA) (4.4. ábra), melyeken többnyire 10 kbit/s és 3,4 Mbit/s közötti átviteli sebesség érhető el (minden egyes órajel impulzusnál egy bit továbbítódik). A részvevők a két vezetékre nyitott draines illetve nyitott kollektoros kimenettel csatlakoznak és a vezetékek elengedett állapotban történő magas logikai szintre történő beállását a felhúzó ellenállások biztosítják.



4.4. ábra Egy példa az I²C hálózat elrendezésére

Az I²C busz legfőbb előnye a cím alapú kommunikáció. Minden egyes buszra kötött eszköz rendelkezik egy 7 (vagy 10) bites címmel, amellyel azonosítani lehet a hálózaton, tehát az

SPI-jal ellentétben nem kell külön engedélyező vezetéket kötni az egyes eszközökhöz. A kommunikáció során a mester vezérli a kommunikációt illetve szolgáltatja az órajelet, ugyanakkor mind a mester, mind a szolga eszközök lehetnek adók illetve vevők is. Az órajel generálásnál lényeges, hogy a szolga eszközök nem generálhatnak órajelet, ugyanakkor a szolga is befolyásolhatja azt, mivel ha nem kész a kommunikációra, akkor lent tarthatja az órajelet, késleltetve ezzel a mestert.

Az SDA adatvezetéket az adatbitek átvitele alatt mindig az aktuális küldő (adatkivitelnél a mester, adatbeolvasásnál a szolga), a nyugtázás ideje alatt pedig a fogadó egység vezérli. Maga az átvitel keretek segítségével történik, melyeket a start illetve a stop fázisok határolnak. Ezen fázisok különlegessége, hogy ekkor az SDA adatvonal az SCL órajel magas értéke alatt vált állapotot. Minden más esetben az SDA adatvonal az órajel magas értéke alatt stabil, és az új értéket az SCL alacsony állapota állítja be. Egy átvitel során előfordulhat ún. ismételt start fázis is, tehát a start-start-stop fázis sorozat is érvényes kereteket jelöl.

Az I^2C esetén lényegében háromféle kommunikáció típusról beszélhetünk. Az egyik esetben mester szeretne adatot küldeni az egyik szolga eszköznek. Ekkor a mester megcímzi a céleszközt majd adatokat küld, és végül lezárja az átvitelt. Egy másik eset, amikor a mester olvasni szeretne a szolga eszköztől. Ekkor a mester megcímzi a céleszközt, majd adatokat fogad a szolga felől, és végül lezárja az átvitelt. Azt, hogy a mester írni vagy olvasni szeretne, a cím utáni első bittel jelzi. A harmadik lehetőség, amikor a mester egy általános hívási címet címez meg, ekkor az üzenet minden szolgának szól.

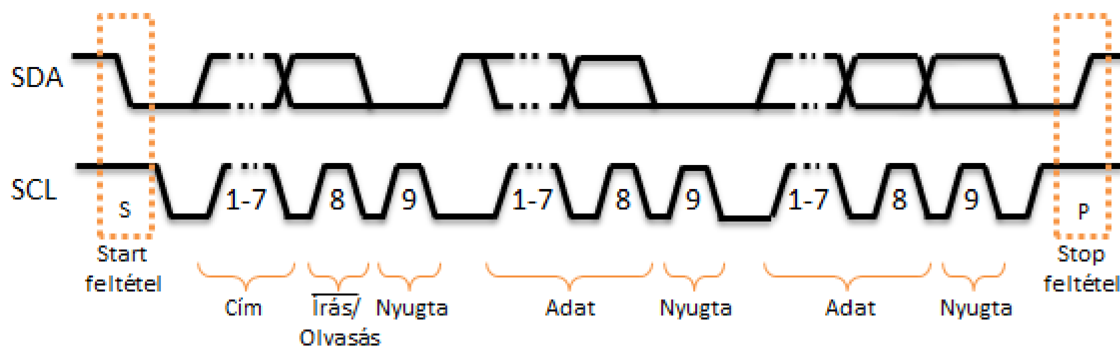
Az I^2C esetében szükség lehet szinkronizációs folyamatra is, mivel az egyes mesterek saját órajelet generálnak. Mivel több mester is lehet, ezért előfordulhat, hogy egyszerre kezdeményeznének kommunikációt. Ilyenkor el kell dönteni, hogy ki fogja vezérelni a kommunikációt, azaz ki nyeri el a kommunikációs jogot. Ez az arbitráció. Ahhoz, hogy az arbitrációs folyamatot le lehessen folytatni, szinkronizálni kell az órajeleket. A szinkronizáció során az SCL vezetéken egy magas-alacsony átmenet az érintett eszközöknél az alacsony periódusuk időzítésének megkezdését eredményezi. Ha egy eszköz órajele alacsonyra váltott, egészen addig alacsony állapotban tartja az SCL vezetéket, amíg az órajele magas periódusához nem ér. Ennek az órajelnek egy alacsony-magas átmenete nem változtathatja meg az SCL vezeték állapotát, ha egy másik eszköz órajele még mindig az alacsony periódusában van, azaz az SCL vezetéket a leghosszabb alacsony periódusú eszköz tartja alacsonyan. Erre az időre a rövidebb alacsony periódussal rendelkező eszközök magas várakozó állapotba kerülnek. Amikor minden érintett eszköz az alacsony periódusa végére ért, az órajel vezeték felszabadul, és magas logikai állapotba kerül. Ekkor nincs különbség az eszközök órajelei és az SCL vezeték állapota között, és minden eszköz elkezdi kiszámolni a magas periódusát. Az első eszköz, amely végzett a magas periódusával, ismét alacsonyra húzza az SCL vezetéket. A fenti eljárásnak köszönhetően az SCL vezetéken egy szinkronizált órajel áll elő, melynek az alacsony periódusát a leghosszabb alacsony periódusú órajel, a magas periódusát az egyik legrövidebb magas periódusú órajel határozza meg.

Ahogy korábban is megemlítésre került, a több mesteres felépítés miatt szükség lehet arbitrációra, akkor, ha kettő vagy több mester próbál információt küldeni, azaz több mester generál egy start feltételt, a start feltétel minimális tartási idején belül. Az arbitráció úgy

zajlik, hogy az első olyan mester elveszti az arbitrációt, amelyik logikai 1-et akar küldeni, miközben a többiek 0-át. Ha a címbitek összehasonlításánál nincs különbség, akkor az arbitráció az adatbitekkel folytatódik a további részeken, így az arbitráció során nincs információvesztés.

Összefoglalva, a kommunikáció az alábbi lépések szerint történik (4.5. ábra):

1. A mester kezdeményezi az adatátvitelt a start fázissal.
2. Kiírja a címet az adatvonalra.
3. A cím utáni egy bites vezérlő jellel megadja, hogy olvasni vagy írni szeretne-e az adott szolgától.
4. A szolga nyugtázza a címének vételét (ACK ,Acknowledge).
5. Ezután következik az írási vagy olvasási ciklus.
6. A mester jelzi az adattranszfer végét a stop fázissal.

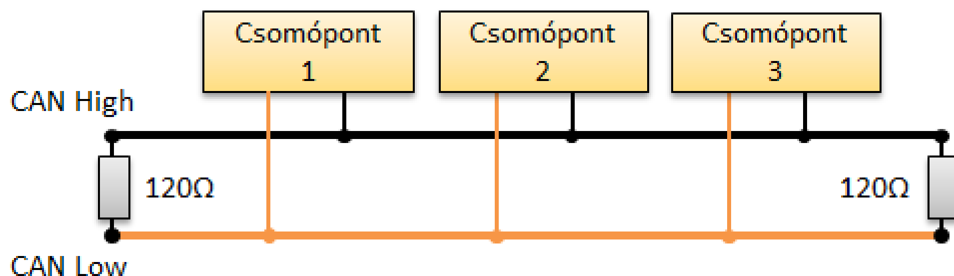


4.5. ábra Egy példa az I²C kommunikációra

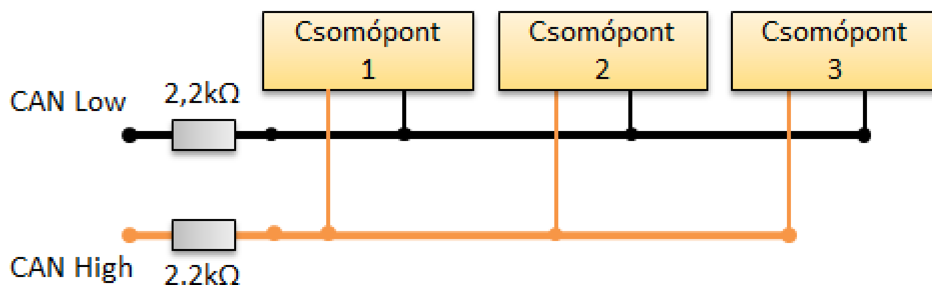
4.1.4 CAN

A CAN egy soros, aszinkron kommunikációs protokoll. A buszra felfűzött összes eszköz (csomópont) egyenrangú, és tetszőleges időpontban kezdhet adni, éppen ezért a protokollnak az I²C-hez hasonlóan veszteségmentes arbitrációt kell biztosítania. Üzenetszórásos jelleggel működik, azaz minden, a hálózathoz csatlakozó eszköz lát minden üzenetet. Fizikai szinten két vezeték használata a kötelező, mivel differenciális feszültségmérésen alapul, és NRZ (nullára nem visszatérő, non-return to zero) kódolást használ. A fizikai jelvezetékek a magas (CAN High), és az alacsony (CAN Low). Ezek mellett többször szoktak még földelést, és néha tápvezetéket is alkalmazni (4.6. ábra és 4.7. ábra).

Kétféle CAN szabványt szoktak használni a nagy (high speed) illetve az alacsony (low speed) sebességű hálózatokat. A nagy sebességű hálózatok 125 kbps - 1 Mbps átviteli sebességet, míg az alacsony sebességű hálózatok 10 kbps - 125 kbps sebességet biztosítanak. Ugyanakkor nem csak a sebességben van köztük különbség. Elsődleges szempont, amiért elkülönítik ezen hálózatokat, hogy az alacsony sebességű hálózatok nagy hibatűrő képességgel rendelkeznek, valamint a nagy sebességű CAN hálózatok esetén többnyire megkövetelik a 120Ω-os lezáró ellenállásokat a megfelelő impedancia elérése miatt.



4.6. ábra Példa egy nagysebességű CAN hálózat felépítésére 3 csomópont esetén



4.7. ábra Példa egy alacsony CAN hálózat felépítésére 3 csomópont esetén

Mivel jelen esetben aszinkron hálózatról van szó, ezért itt is fontos, hogy minden egyes csomópont a megfelelő módon legyen beállítva, azaz ugyanarra a hálózati sebességre (azonos bitidőkre) legyen bekonfigurálva.

A CAN hálózatok összetettsége miatt (mivel nagyon sok csomópont küldhet nagyon sokféle üzenetet) adatbázisokat szoktak létrehozni annak a leírására, hogy melyik csomópont mikor milyen üzeneteket küld és ezeket az üzeneteket ki fogja értelmezni. Ezen adatbázisok többnyire az üzenetekben található adatmező felosztását is tartalmazzák, ugyanis az adatmezők értelmezéskor azok tetszőlegesen feloszthatóak kisebb részekre, mint például a négy keréksebességet el lehet küldeni egy üzenet adatmezőjében egyszerre.

4.1.4.1 Az adatküldésre szolgáló üzenetek felépítése

A CAN esetében az üzenetek már összetettebbek, mint az eddig ismertetett kommunikációs protokollok esetén. Alapvetően kétféle üzenet típust kell megkülönböztetni: a hagyományos és kiterjesztett üzeneteket. A hagyományos üzenetek esetében 11 bit, míg a kiterjesztett üzeneteknél 29 bit áll rendelkezésre az üzenet azonosítókhoz.

CAN 2.0A (Hagyományos)

SOF	ID	RTR	IDE	R0	DLC	DATA	CRC	ACK	EOF	IFS
1	11	1	1	1	4	0...8	16	2	7	3...

CAN 2.0B (Kiterjesztett)

SOF	ID A	SRR	IDE	ID B	RTR	R1	R0	DLC	DATA	CRC	ACK	EOF	IFS
1	11	1	1	18	1	1	1	4	0...8	16	2	7	3...

4.8. ábra Az adat típusú CAN üzenetek felépítése

A hagyományos üzenet esetében a következők szerint épül fel egy üzenet:

- Minden egyes üzenet egy start bittel (SOF (start-of-frame)) indul, mely egy domináns bit, amit minimum 11 recesszív bit előz meg.
- Ezután következik 11 bit, ami az üzenet azonosítóját tartalmazza (ID, identifier). A CAN esetében lényeges, hogy nem a hálózathoz csatlakozó eszközöknek, hanem az üzeneteknek van azonosítója, ami meghatározza az eszközök számára annak tartalmát, valamint a prioritását is. Ugyanis a CAN esetén, ha két eszköz, más néven csomópont egyszerre próbál meg adni, akkor az I^2C-hez hasonló arbitráció kezdődik. Mindkét csomópont elkezdi kiküldeni a saját üzenetét. Minden egyes bit után, amit kiküldenek vissza is olvassák a hálózaton kint levő értéket. Amint az egyik csomópont azt detektálja, hogy ő recesszív bitet helyezett ki a hálózatra, de domináns bitet olvasott vissza, abbahagyja az üzenetküldését. Ebből következik, hogy az a csomópont nyeri el az adás jogát, aki alacsonyabb azonosítóval rendelkező üzenetet szeretne küldeni, mivel a CAN esetében a domináns szint a logikai 0-nak felel meg. Egy jól megtervezett hálózat esetén elegendő az azonosító mező az arbitrációhoz, mivel egy adott azonosítóval rendelkező üzenetet nem küldhet két csomópont, ám ha ilyen előfordulna, akkor az arbitráció folytatódik az üzenet további részén.
- Az azonosító mezőt követi a RTR (remote transmission request) bit, mely azt hivatott jelezni, hogy adatkérést tartalmaz-e az üzenet. A legtöbb esetben a CAN hálózaton adatszórás jellegű kommunikáció zajlik, tehát egy bizonyos időközönként az adott csomópont megpróbálja kiküldeni az adott információt, mondjuk egy vagy több szenzornak a jelét. Ugyanakkor lehetőség van adat kérésére is. Ez az adatkérő üzenetek segítségével történik. Felépítésben az adat üzenet illetve adatkérő üzenetek szinte teljesen megegyeznek egy fontos kivétellel, hogy az adatkérő üzeneteknek nincs adatmezőjük.
- Az RTR bitet követi a kiterjesztett üzenetet jelző bit (IDE, identifier extension). Hagyományos üzenetek esetében ez mindig 0, mivel nem tartalmaz kiterjesztett azonosítót.
- Az ezeket követő bit egy helyfenntartó bit, melynek mindig nullának kell lennie.
- Ezután következik négy bit (DLC, data length code), mely azt tartalmazza, hogy hány bájtnyi adatot tartalmaz az üzenet. A CAN üzenetekben alapértelmezetten 0-tól 8 bájtig terjedő adatmennyiséget lehet továbbítani.
- A DLC-t követi maga az adat. Az adat tetszőleges információkat tartalmazhat, ezeket már csak a csomópontok értelmezik.
- Az adatot egy 15 bites CRC (cyclic redundancy check) ellenőrző összeg követ, illetve még egy 1 bites CRC lezáró, ami mindig 1-es értéket tartalmaz.
- Ezután egy nyugta mező következik, mely két bitből áll. Az első bitnek az a feladata, hogy jelezze az adó felé, hogy valaki vette az üzenetet, ugyanis bármelyik csomópont, amelyik veszi az üzenet korábbi részét az domináns állapotba, azaz nullára állítja a hálózaton levő jelszintet. Ennek azért van szerepe, hogy az adó tudja, hogy vette-e valaki az üzenetet. A második bitje a mezőnek egy recesszív elválasztó bit.
- Ezeket követi az üzenetet lezáró hét recesszív bit (EOF (end-of-frame)), majd ezt legalább három további recesszív bit, ami az üzenetek elválasztására szolgál (IFS

(interframe space)). Ezeket összeadva valamint hozzászámolva a nyugta elválasztó bitet, megkapható a start bitnél említett 11 recesszív bit.

A kiterjesztett üzenetek nagyon hasonlítanak a hagyományos üzenetekhez, a lényegi eltérés az azonosító mező kiterjesztésében van. A kiterjesztett üzenetek esetében a 29 bites azonosító első 11 bitjét a hagyományos címnek fenntartott azonosító mező tartalmazza. Ezt követően az RTR mező eredeti helyén egy SRR (substitute RTR bit) bit található, melynek recesszív értékűnek kell lenni, amelyet az IDE bit követ, ami ebben az esetben egyes értéket fog felvenni. Az IDE mezőt az azonosító fennmaradó 18 bitje követi. Ezt követően következik az RTR bit, majd egy 1 bites helyfenntartó. Innentől kezdve már megegyezik a hagyományos és kiterjesztett üzenetek felépítése.

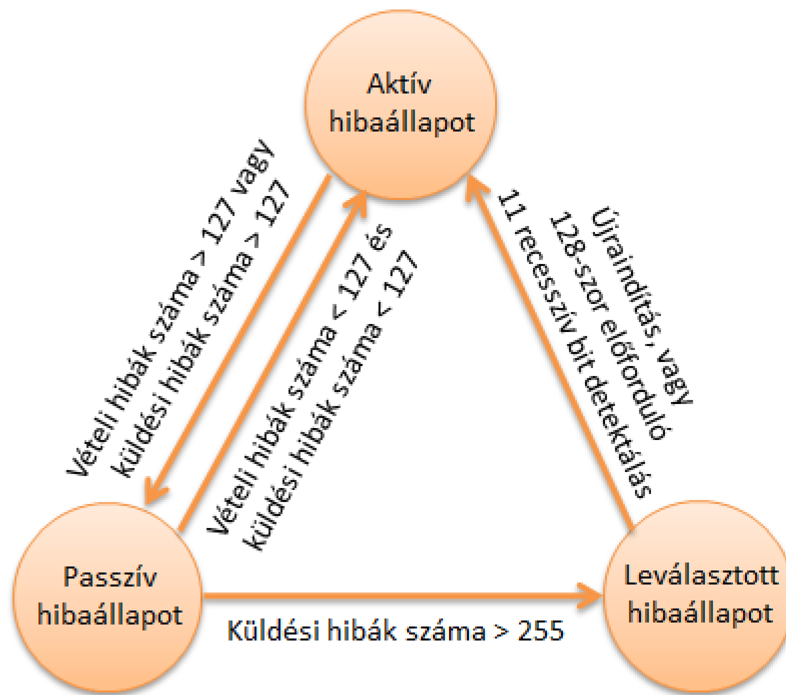
Az üzenetek felépítésénél megemlítésre került, hogy 11 recesszív bitet követ egy start bit, így minden eszköz pontosan tudja, hogy új üzenet kezdődött. Belegondolva, hogy a 11 recesszív bit a nullás azonosítóval rendelkező üzenet esetén is kijöhet. Ezt azonban a CAN protokoll orvosolja a bitbeszúrás (bit stuffing) nevezetű eljárással. A módszer lényege, hogy minden öt azonos bit után (beleértve a beszúrt biteket is) beszúr egy ellentétes bitet. Természetesen ez az eljárás csak abban az esetben működőképes, ha a csomópontok megfelelően vannak beállítva (bit idők, stb.). Maga az eljárás nem csak az üzenetek felismerését, hanem az üzenetek közbeni szinkronizációt is elősegíti.

4.1.4.2 Hibakezelés

A CAN protokoll a hibakezelésre vonatkozóan is tartalmaz leírást. Három hibaállapotot definiál minden egyes csomóponthoz:

- Aktív hibaállapot (error active): Ebben az állapotban, ha hiba történik, akkor a csomópont azt feltételezi, hogy azt nem ő okozta.
- Passzív hibaállapot (error passive): Ebben az állapotban, ha hiba történik, akkor a csomópont azt feltételezi, hogy valószínűleg helyi meghibásodás történt, de még nem olyan súlyos a helyzet, hogy le kelljen válnia a buszról.
- Leválasztott hibaállapot (bus off): Ebben az állapotban a csomópont feltételezi a hibás működést, melyet kellően súlyosnak gondol ahhoz, hogy leválassza magát a buszról.

A különböző állapotok között minden egyes csomópont a saját hibaszámlálója segítségével képes lépkedni. Ahogy növekszik az egymást követő vételi illetve küldési hibák száma, úgy lépteti át magát az adott csomópont az egyik állapotból a másikba (4.9. ábra).



4.9. ábra CAN csomópont hibaállapotok közötti átmenete

A hibakezeléshez további három üzenet típus tartozik. Bármilyen küldés vagy fogadás során fellépő hiba egy hibaüzenetet generál, ami szándékosan megsérti a bitbeszúrás szabályait, ezzel kényszerítve az adó csomópontot az újraküldésre. Az aktív és a passzív hibaállapotokhoz különböző üzenetek tartoznak, ugyanakkor mindkettő két tagból áll: egy hibajelző (Error Flag) és egy hibahatároló (Error Delimiter) mezőből.

- Az aktív hibaüzenet a kezdeti hiba észlelésekor kerül kiküldésre úgy, hogy egy vagy több aktív hibaállapotban levő csomópont azonnal megszakítja a kommunikációt (kivéve CRC hiba esetén). Ezt úgy teszik, hogy domináns biteket helyeznek a buszra. Az első 6 domináns bit alkotja az aktív hibajelző részt, vagyis az aktív hibaüzenet első mezejét. Ezt követően recesszív bitet kezd el adni a csomópont.

  Minden olyan aktív hibaállapotban lévő csomópont, amely a kezdeti hibát nem érzékelte, legkésőbb az aktív hibajelző mező 6. domináns bitjénél hibát fog generálni, ugyanis ezen a ponton a bitbeszúrás szabálya sérül. Így legrosszabb esetben újabb 6 domináns bit fog a CAN buszon megjelenni, ezért ez a szakasz az aktív hibajelzők szuperpozíciója. E szakasz hossza ismeretlen, 0-6 bit hosszúságú lehet. Ha 0 bit hosszúságú, akkor a kezdeti hibát egyszerre észlelte az összes aktív hiba állapotú csomópont.

  Ahogy a CAN buszt figyelik a csomópontok, az aktív hibajelző 6 domináns bitet követően - egy idő után - recesszív bitet fog visszaolvasni minden csomópont, melyet követően még 7 recesszív bitet sugároznak a csomópontok. Az aktív hibaüzenet záró része tehát a 8 recesszív bitből álló hibahatároló mező. Ezzel a módszerrel lehetségessé válik egy csomópont számára, hogy érzékelje, hogy vajon ő volt-e az első, aki hibajelzést küldött, azaz elsőként észlelte a hibát. A hibás csomópontok elszigetelésénél fontos ez a mechanizmus.

Az aktív hibaüzenet végén a busz ismét kész adathordozó üzenet továbbítására. Így az a csomópont, amelyiknek adása meg lett szakítva, megkísérelheti az el nem küldött üzenet újraküldését.

- A passzív hibaüzenetek esetén egy csomópont passzív hibaállapotban van, melyben passzív hibaüzenet küldésére képes. A Passzív hibaüzenet első fele a Passzív hibajelző, amely 6 recesszív bitből áll. Ennek csak akkor van hatása, ha a passzív hibaállapotú csomópont a megfelelő helyen kezdi el a passzív hibaüzenet küldését. Azt a passzív hibajelzőt, amely arbitrációs mezőben, valamint kevesebb, mint hat bittel a CRC sorozat vége előtt kezdett adni, nem érzékeli a többi csomópont.

  Tehát ha egy nem buszbirtokló csomópont próbál Passzív hibajelzést adni, akkor annak nem lesz hatása a hálózat többi csomópontjára.

  A „Hiba passzív" állapotú csomópontoknak mindig ki kell várni a 6 azonos értékű recesszív passzív hibajelző bitet a hiba detektálása után, hogy befejezettnek tekinthessék a hibajelzésüket, melyet a hibahatároló 8 recesszív bit zár le, megegyezően az aktív hibaüzenettel.

- A túlcsordulás üzenetnek ugyanolyan formátuma van, mint az aktív hibaüzenetnek, ugyanakkor hatására nem szükséges az előző üzenet újraküldése. Egy csomópont több különböző esetben is küldhet túlcsordulás üzenetet. Ilyen például, ha a fogadó csomópont késleltetni akarja a következő üzenet fogadását, ha a fogadó csomópont az üzenetek közötti mező első két bitjén domináns bitet érzékelt, illetve ha a hiba-, vagy túlcsordulás-határoló mező utolsó bitjén domináns bitet érzékelt.

  Túlcsordulás üzenetet csakis „Hiba aktív" állapotú fogadó csomópont küldhet, abban az esetben, ha nem kész a következő üzenet fogadására. Egymás után maximum kettő küldhető, és csupán az üzenet utáni szünetben fordulhat elő.

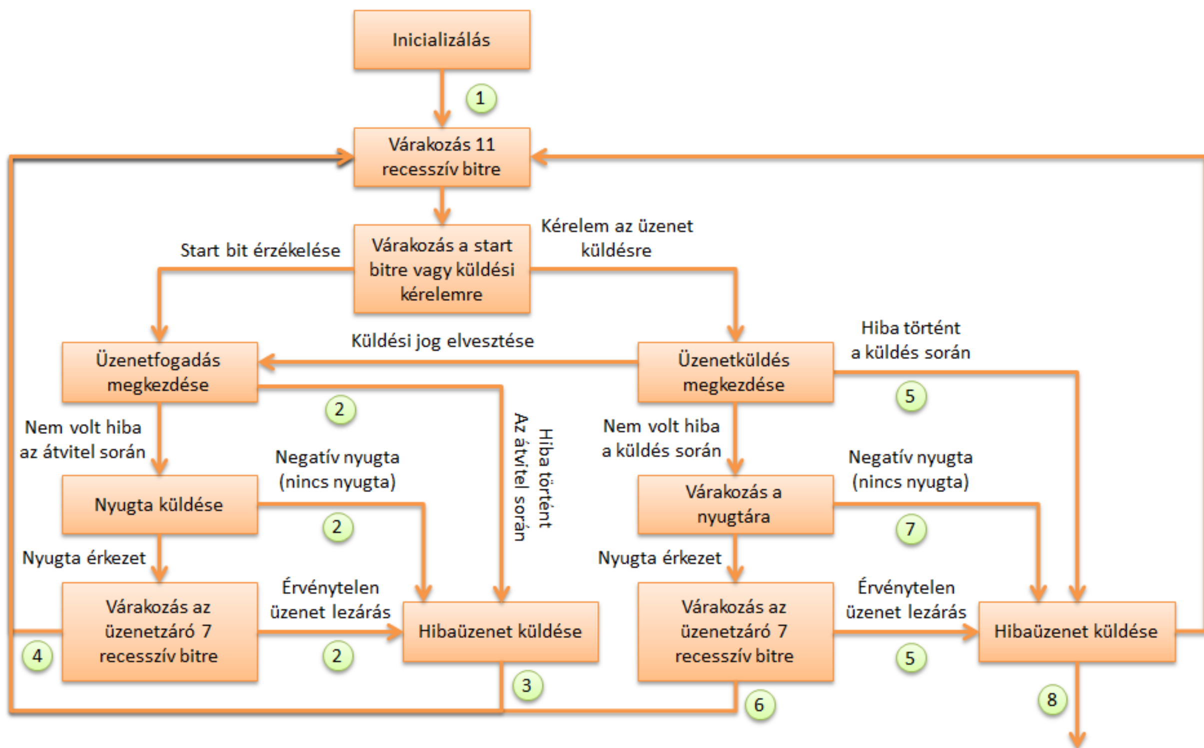
4.1.4.3 A CAN működési diagramja

A CAN csomópontok alapvető működése felrajzolható egy diagram segítségével (4.10. ábra). Az ábrán az egyes számozott pontok bizonyos műveletek elvégzését jelentik:

1. Inicializálás, hibaszámlálók és működési paraméterek alapértelmezett állapotba állítása.
2. Vételi hibák számának rögzítésére szolgáló számláló növelése eggyel.
3. Vételi hibák számának rögzítésére szolgáló számláló növelése nyolccal, hogy ha a vevő elsőnek állítja be a hiba jelzésére szolgáló jelzést.
4. Vételi hibák számának rögzítésére szolgáló számláló csökkentés eggyel az üzenet sikeres vételekor.
5. Küldési hibák számának rögzítésére szolgáló számláló növelése nyolccal, hogy ha hiba történt az átvitelkor.
6. Küldési hibák számának rögzítésére szolgáló számláló csökkentés eggyel az üzenet sikeres átvitelekor.
7. Küldési hibák számának rögzítésére szolgáló számláló növelése nyolccal. Abban az esetben, ha a küldési hibákhoz tartozó számláló nagyobb, mint 127, valamint ha a

passzív hibák jelzésére szolgáló jelzés recesszív maradt, akkor nem kell növelni az értéket.
8. Ha a küldési hibák számának rögzítésére szolgáló számláló nagyobb, mint 255, akkor a csomópontot le kell választani a buszról.

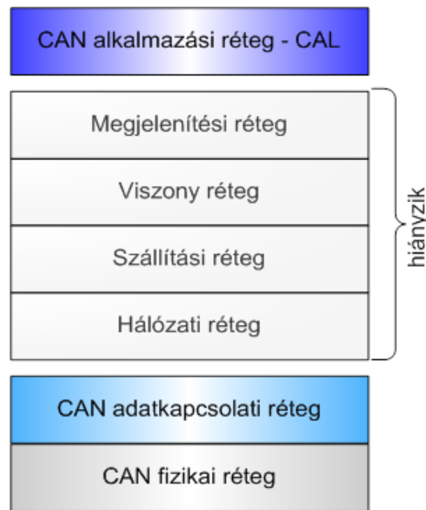
A helyes működési elvhez feltételezni kell, hogy a vételi hibák számát rögzítő számláló nem növekedhet 128 fölé, valamint sem ez, sem a küldési hibák rögzítésére szolgáló számláló nem csökkenhet 0 alá.



4.10. ábra A CAN csomópontok működési diagramja

4.1.5 CANOpen

A CAN specifikációja a bitek fizikai továbbításával, azok keretezésével, valamint alapvető, általános kérdésekkel foglalkozik, tekintve, hogy a fizikai és az adatkapcsolati réteg definíciójára összpontosít. Ugyanakkor ipari felhasználási területeken szükséges volt az alapvető specifikációban foglaltakat meghaladó szolgáltatások szabványosítására is, mint például az alkalmazás-specifikus finomításokra és bővítményekre. Ennek érdekében definiálta a CiA a CAN alkalmazási rétegét (CAL). Ez az új réteg tulajdonképpen a kommunikációt végrehajtó rendszer és az applikáció közötti interfésznek tekinthető (4.11. ábra).



4.11. ábra Az alkalmazási réteggel kibővített CAN modell

A CAN alkalmazási rétege számos szolgáltatást nyújt, többek között saját üzenetformátumokat és üzenetobjektumokat definiál, megadva ezek segítségével az egyes eszközök nyújtotta funkciók elérésének módját. Az üzenetekbe kerülő adatok kódolására is definiál megkötéseket, saját adattípusokat alkalmaz, egységesebbé téve ezzel a szállítást.

A CAL számos általános szolgáltatás definiál, ugyanakkor sokszor még mindig nehezen kezelhető. A könnyebb kezelhetőség elérése érdekében kezdődött meg a CANopen fejlesztése, mely a CAL-re épül, s annak szolgáltatásait használva jól kezelhető, egységes felületet nyújt az ipari alkalmazásokhoz, lefedve a lehetséges eszközök túlnyomó részét. Alkalmas mind elosztott, mind központosított szabályozási architektúra megvalósítására, paraméter fel- és letöltésre. A buszra fűzött eszközökhöz funkció alapján standard kapcsolódási, kezelési felületet kaptak, a rendszer a legtöbb mikrokontroller esetében működőképes, és megőrizte a CAN összes jellemzőjét.

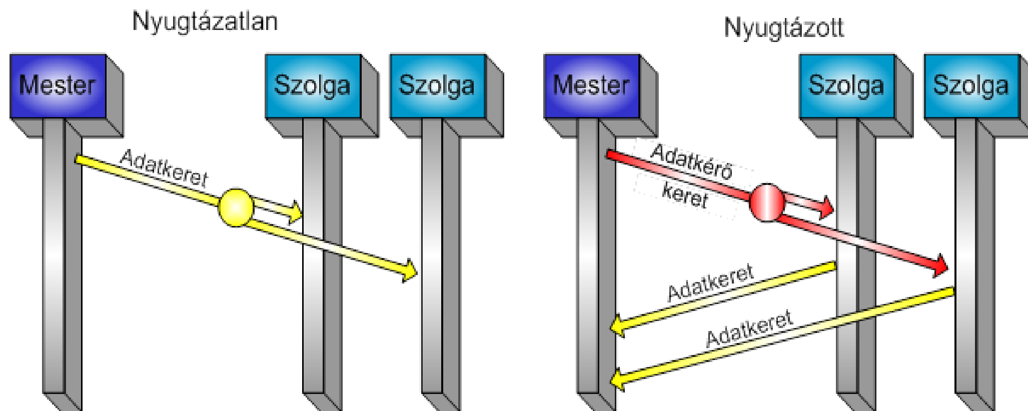
A CANopen valamennyi csomópontja tekinthető egy összetett, sokcélú eszköznek, melynek feladata a kapcsolat fenntartása a CAN busz és az adott eszköz által végzett művelet között. A modellben interfészként szereplő objektumkönyvtár (Object Library) az adott eszköz összes adatát, paraméterét tartalmazza indexelt formában. Ezt a táblázatot a kommunikációs objektumokon (Communication Object) keresztül írni és olvasni lehet a busz felől, így fejtve ki hatását az adott alkalmazásra az alkalmazási objektumokon (Application Object) keresztül. A kommunikációs objektumok mindegyike egy-egy specifikus kommunikációs feladatért felelős, azaz egy-egy meghatározott üzenet küldéséért vagy fogadásáért. Ezek között adatszállítók és a rendszer általános működését irányítók is megtalálhatók. Ez utóbbiak irányíthatják az eszközök állapotváltásait is, a belső állapotgépnek adva parancsokat (Network Management = NMT). Vannak olyan adatszállító üzenetek is, melyek közvetlenül az objektumkönyvtár érintése nélkül érik el az alkalmazási objektumokat (Process Data Object = PDO).

Kihasználva a CAN által biztosított nagy rugalmasságot, a CANopen három alapvető kommunikációs módot definiál:

- Mester-szolga (Master-Slave)

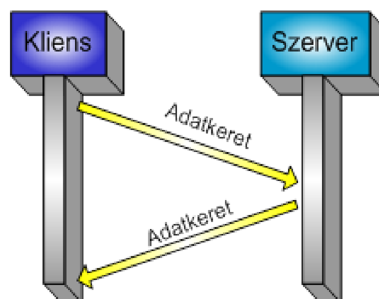
- Kliens-szerver (Client-Server)
- Gyártó-fogyasztó (Producer-Consumer)

A mester-szolga kommunikációs modellben egy kitüntetett csomópont, a mester lép interakcióba a hálózat összes többi csomópontjával, a szolgákkal. A szolgáltatás lehet nyugtázott és nyugtázatlan, általában hálózatvezérlésre használják (NMT objektumok) (4.12. ábra).



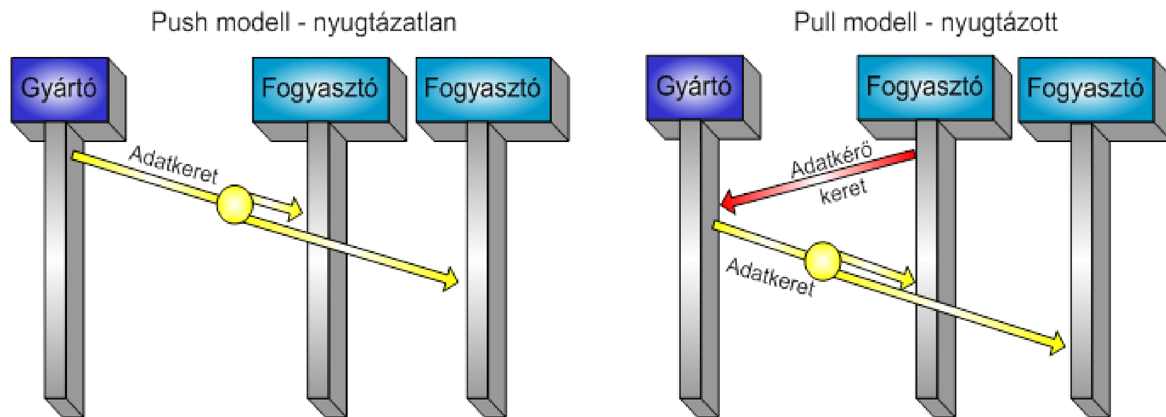
4.12. ábra A Mester-szolga kommunikációs modell

A kliens-szerver kommunikációs modellben egy csomópont, a kliens lép kapcsolatba egy másik csomóponttal, a szerverrel, ahol adatot ír vagy olvas. Eszközkonfiguráláshoz használják, s ennek megfelelően csak nyugtázott formája létezik, ahol a válasz a művelet eredményét jelzi a kliensnek (Service Data Object = SDO objektumok) (4.13. ábra).



4.13. ábra A Kliens-szerver kommunikációs modell

A gyártó-fogyasztó modell hasonlít a mester-szolga megoldáshoz, mivel itt is egy kitüntetett csomópont, a gyártó küld adatot több más csomópontnak. Ám azok száma nem kötött, illetve az is előfordulhat, hogy elküldött csomagjainak nincs vevője. Tipikus felhasználása a műveleti adatok valósidejű (real-time) cseréje. A CANopen terminológia ennek a kommunikációs modellnek megfelelő objektumokat műveleti objektumoknak (PDO-nak) nevezi. Ha a gyártó kérés nélkül küldi az adatot a fogyasztóknak, azt nevezik push (toló) modellnek. Azonban a protokoll lehetőséget biztosít arra, hogy bármely fogyasztó bármely időpillanatban üzenetek segítségével kérje az adatot a gyártótól. Ez esetben pull (húzó) modellről lehet beszélni. A nyugtázás szempontjából az előbbi a nyugtázatlan, az utóbbi a nyugtázott üzenetcsere osztályába tartozik (4.14. ábra).



4.14. ábra A gyártó-fogyasztó kommunikációs modell

További speciális objektumok:

- A szinkronizáló objektum (Synchronisation Object - Sync) a CANopen hálózatra kötött eszközök szinkronizálását végzi.
- A vészhelyzet objektum (Emergency Object) a hibakezelés során alkalmazzák a vészhelyzet jelzésére.
- Az időbélyeg objektum (Time Stamp Object) a teljes hálózat egységes időzítésére használható.

4.1.5.1 A szolgáltatási objektumok (SDO)

A szolgáltatási objektumok elsősorban az eszközök konfigurálásához használhatók. Az eszközök objektumainak teljes skálájához hozzáférhető, mely objektumok egymástól nagymértékben különbözőek lehetnek. Emiatt nagyobb rugalmasságot mutatnak, s akár 8 byte-nál hosszabb üzenet továbbítását is lehetővé teszik CAN üzenetek láncával. Ennek megfelelően több átviteli módot ajánl a CANopen az SDO-átvitel számára:

- Gyorsátvitel (Expedited Transfer): kis adatok szállítására kiélezett átviteli mód, melynek implementálását a CANopen specifikációja kötelezővé teszi valamennyi eszköz számára. Használatára is kötelezi őket minden, 4 byte-ot meg nem haladó adat esetén.
- Szegmentált átvitel (Segmented Transfer): az átvitel általánosan alkalmazott módja, mely minden esetben alkalmazható. Implementálása akkor kötelező, ha az eszközben lehetőség van 4 byte-nál hoszabb objektumok implementálására is.
- Blokkátvitel (Block Transfer): a hosszú üzenetek átvitelére optimalizált átviteli forma, melynek implementálása teljes mértékben opcionális.

Mivel a kliens célja kétféle lehet, írás vagy olvasás, a kliens-szerver kapcsolat két különböző szolgáltatás nyújt: feltöltést (Upload) és letöltést (Download). Az átviteli módok bármelyike esetén a kliens a kapcsolat létesítője, viszont a kapcsolat lebontását mind a szerver, mind a kliens kezdeményezheti.

4.1.5.2 A műveleti objektumok (PDO)

A szolgáltatási adatobjektumokkal ellentétben a műveleti adatobjektumok rövid blokkjai direkt módon férnek hozzá az alkalmazási objektumokhoz, az objektumkönyvtár

megkerülésével. Erre a lehetőséget az adja meg, hogy előre rögzített PDO-leképezési szabályok alapján a PDO bitjeinek jelentése ismert mind a küldő, mind a fogadó számára. Emiatt a műveleti adatobjektumok alkalmasak valós idejű feladatok végrehajtására, gyors kommunikációra, melyet erős prioritású azonosítókkal támogat meg a rendszer. A CANopen azonban itt is flexibilis: az arra alkalmas eszközök esetén a PDO-k bitjeinek leképezése SDO-kon keresztül módosíthatók. Ugyanez igaz a bennük foglalt adatok kódolására, adattípusaira is. A gyártó oldalán az úgynevezett továbbítási PDO-t (Transmit PDO) implementálják, míg a fogyasztó oldalán a vételi PDO-t (Receive PDO). Minden egyes PDO számára ki kell osztani egy azonosítót, melyhez a CANopen előre definiált sémát nyújt. Ez alapján négy vételi és négy továbbítási PDO-val rendelkezhet egy CANopen eszköz.

Mivel a PDO-k az alkalmazás saját adataihoz vannak rendelve, mely adatok állandó feldolgozás alatt állnak, több módot ajánl a CANopen ezen adatok megszerzésére.

- Adatkérő (Remotely Requested): Ez az a mód, melyet az olvasási szolgáltatás használ, ahogy azt már tárgyaltuk. Egy adatkérő keret vétele váltja ki a PDO elküldését.
- Időzített (Timer Driven): A PDO elküldése egy bizonyos idő elteltével következik be. Az írási szolgáltatás használja ezt a módot.
- Eseményvezérelt (Event Driven): Egy, az adott eszközön belüli meghatározott esemény bekövetkezte indítja útjára a PDO-t. Az írás használja ezt a módot. Ez a gyakorlatban leginkább alkalmazott mód, ám nem kötelező egymagában használni: egy PDO egyszerre lehet erre a módra, s az időzített, vagy az adatkérő mód valamelyikére konfigurálva.

A CANopen több módot kínál a PDO-átvitel kezdeményezésére, valamint magára az átvitelre is több lehetőséget definiál:

- Szinkron átvitel (Synchronous Transmission) során a PDO-k küldése periodikus, mivel az egy periodikusan generált szinkronizáló objektum (Synchronisation Object) vételének hatására történik. Ezt egy másik eszköz állítja elő, s általában több eszköz veszi, melyek így egyszerre küldik el adataikat, ill. kezdik el feldolgozni az egy periódussal korábban kapottakat. Értelemszerűen ez a típusú átvitel eseményvezérelten működik.

  Ugyanakkor nem kötelező egyszerűen csak a szinkronizáló objektumhoz hangolni a PDO átvitelt, ezt lehet tovább finomítani. Ciklikus szinkron az átvitel, ha csak egy bizonyos számú vett szinkronizáló objektum után történik meg a PDO elküldése. Ciklikus pedig abban az esetben, ha a szinkronizáló üzenet megérkezte után, csak egy bizonyos esemény bekövetkeztekor történik meg a PDO átvitele.

- Aszinkron átvitel (Asynchronous Transfer) esetén a PDO küldését egy adott esemény váltja ki, mely természetesen a szinkronizáló objektumtól független. Ha ezt az eseményt egy alkalmazási objektum generálja, a kezdeményezés egyértelműen eseményvezérelt; míg ha azt egy kommunikációs objektum generálja, időzített és adatkérő módban is működhet az átvitel.

Mivel a CANopen a CAL rétegre épül, ami az alkalmazás rétegnek felel meg az OSI modellben, így a CANopen-t az erre épülő felhasználói rétegként szokták emlegetni.

4.1.6 LIN

A LIN (Local Interconnection Network) lényegében a CAN hálózatok kiegészítőjeként jött létre, annak érdekében, hogy ahol lehet, ott csökkenteni lehessen a hálózat összetettségét valamint költségeit. Elsősorban a nagyobb központi számítógépek, valamint a szenzorok illetve aktuátorok közötti kommunikációra lett megtervezve, ahol nincs szükség nagy sávszélességre.

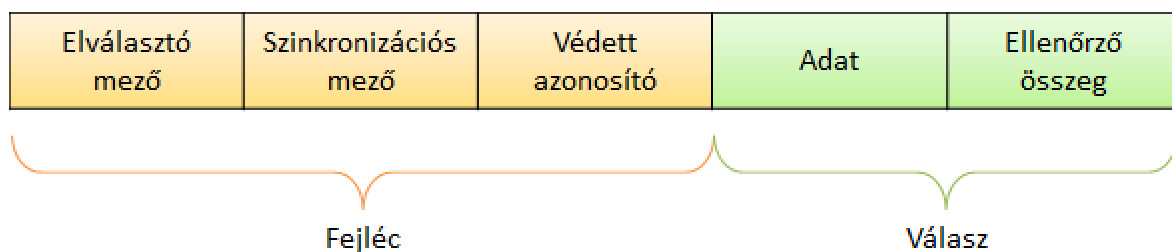
LIN szabványnak több egymást követő verziója van, melyeket a mai napig használnak. Ezek a LIN 1.x és a LIN 2.x fő verziók közé sorolhatóak.

Általánosságban elmondható, hogy a LIN egy aszinkron soros protokoll, mely egy mester több szolga (maximum 15 szolga illeszthető egy vonalra) kapcsolatot definiál, alacsony sebességű kommunikációhoz (általában 1 és 20 kbit/s közötti átviteli sebesség érhető el). A kommunikációhoz egy vezetékre van szükség, ugyanakkor ezen vezeték mellett sokszor szoktak még földelés és tápvezetéket is biztosítani.

Fontos megjegyezni, hogy a CAN-hez hasonlóan a LIN esetében is szoktak hálózat-, illetve kommunikációt leíró adatbázist alkalmazni, mely leírja az üzenetek tartamát valamint, hogy mely üzenethez kinek és mikor kell biztosítania az adatot. Itt a fő különbség a hagyományos CAN adatbázisokhoz képest, hogy egy adatbázis többféle ütemezést (schedule) is tartalmazhat, melyek között működés közben váltani lehet. Ez lehetővé teszi, hogy eltérő körülmények esetén eltérő időközönként más-más üzenetek jelenjenek meg a hálózaton. A különböző ütemezések között a mester tud váltani, mivel ő vezérli az adatok küldését.

4.1.6.1 A LIN üzenetek felépítése

A LIN üzenetek több részre bonthatóak, melyeknek szinte mindegyike érvényes UART üzenetek formájában továbbítódik (4.15. ábra).



4.15. ábra A LIN üzenetek fő részei

Ez alól az egyetlen kivétel az üzenet kezdetének jelzésére szolgáló elválasztó mező (break field). Ez a mező szándékosan megsérti az UART protokollnál leírt üzenetformátumot, így jelezve az üzenet küldésének a kezdetét. Magának a protokollnak a megsértése abból áll, hogy a mester előbb 13 - 26 darab domináns, majd 1 - 14 darab recesszív bitet küld. A leglényegesebb a domináns bitek magas száma. Erre azért van szükség, mert a LIN létrehozásakor az olcsóság volt az egyik elsődleges szempont, így nem követel meg nagy pontosságú időzítést (órajelet) a szolgák oldalán, ezért biztosítani kell, hogy a szabványban megengedett az ideális órajeltől akár 14%-kal is eltérő órajelet alkalmazó szolga egységek is megfelelően detektálják a protokoll megsértését.

Az elválasztó mező után kezdődik meg a szinkronizáció (sync field). Ahogy már korábban is említésre került, a szolga egységek órajelei között nagy eltérések lehetnek, ezért szükséges, hogy a nagy pontosságú órajel generátorral rendelkező mester egységhez, pontosabban annak bitidejéhez szinkronizáljanak. Erre szolgál a szinkronizációs mező. Ebben a részben a mester egy 0x55-öt tartalmazó UART üzenetet küld ki a hálózatra, ami 01010101b értéknek felel meg binárisan, azaz egymást követik egyesek és nullák felváltva. A szolga egységek a lefutó éleket figyelve meg tudják határozni a mesterhez tartozó bitidőt.

A szinkronizációt követően következik az üzenetazonosító, mely egy 6 bites üzenetazonosítót valamint 2 paritás bitet tartalmaz, ahol a paritás bitek az azonosítóból számíthatóak a következőek szerint:

- P0 = ID0 XOR ID1 XOR ID2 XOR ID4
- P1 = INV(ID1 XOR ID3 XOR ID4 XOR ID5)

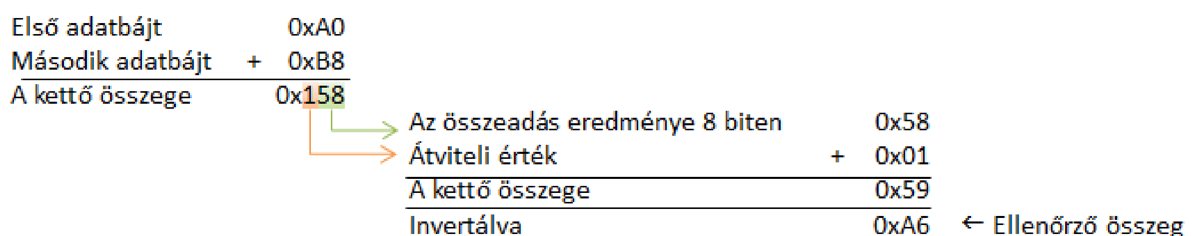
Az azonosítót és a hozzá tartozó két paritás bitet együtt védett azonosítónak (PID (protected ID)) szokás nevezni.

Az eddigi mezőket, azaz az elválasztó mezőt, a szinkronizációs részt valamint a védett azonosítót mindig a mester küldi és ezeket együtt fejlécnek szokás nevezni. Ezután következik a válasz szakasz, melyet mind a mester, mind a szolgák küldhetnek.

A válasz szakasz első fele az 1-8 UART üzenetből álló adatmező, melyből látható, hogy a LIN esetében maximum 8 bájtnyi adatot tartalmazhat egy üzenet.

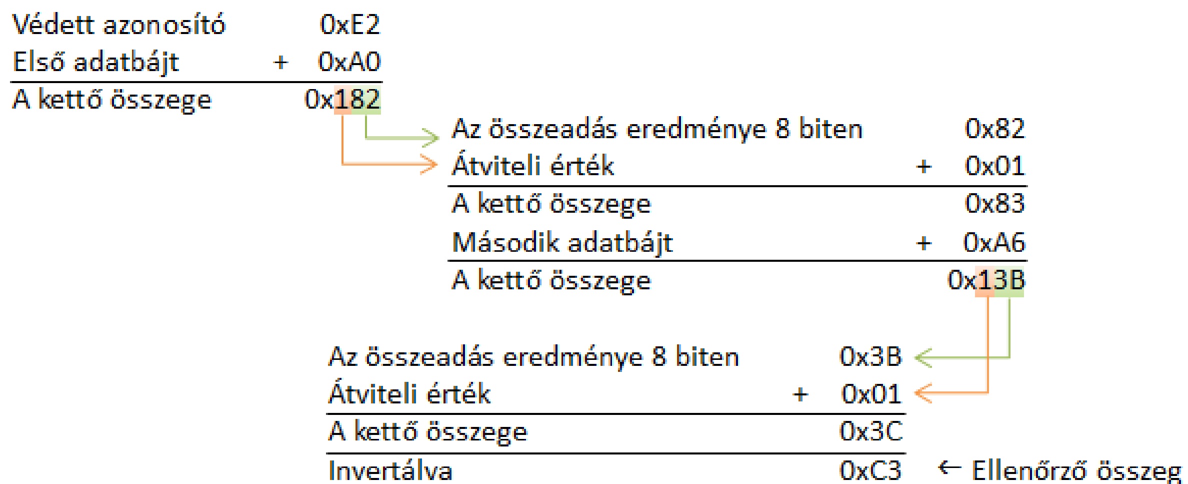
Az adatokat a hozzájuk tartozó ellenőrző összeg követi. Az ellenőrző összeg számítására kétféle módszer létezik: egy hagyományos illetve egy kiterjesztett számítási mód. Az előbbit a LIN 1.x valamint a 60-63-as azonosítóval rendelkező LIN 2.x verziójú üzenetek esetében szokták alkalmazni, míg az utóbbit 0-59-es azonosítóval rendelkező LIN 2.x verziójú üzenetek esetében használatos.

Az ellenőrző összeg úgy számítandó, hogy a számításban résztvevő értékeket egymás után össze kell adni 8 biten, figyelembe véve az átvitt értéket is, majd a végösszeget invertálni kell. A hagyományos módszernél csak az adatbájtok vesznek részt a számításban, míg a kiterjesztett esetében a védett azonosító is. Példaképpen egy 2 bájt adatot tartalmazó üzenet ellenőrző összege a következő féleképpen számítandó a hagyományos módszerrel (4.16. ábra).



4.16. ábra Hagyományos ellenőrző összeg számítása két adatbájt esetén

A hagyományos ellenőrző összeg számításánál alkalmazott adatokon elvégezhető a kiterjesztett módszerhez kapcsolódó számítások is (4.17. ábra).



4.17. ábra Kiterjesztett ellenőrző összeg számítása két adatbájt esetén

Ezek alapján látszik, hogy a kommunikációt teljes egészében a mester vezérli, tehát ő mondja meg, hogy mikor milyen adatot vár, amelyre a szolgák válaszképpen elküldhetik a várt adatot a hozzá tartozó ellenőrző összeggel együtt, illetve a mester is küldhet adatot a szolgák felé, ekkor a válasz szakaszt is a mester tölti ki. Ugyanakkor az is megfigyelhető, hogy a védett azonosító, valamint az ellenőrző összeg segítségével hibadetektálás is elvégezhető.

Fontos megjegyezni, hogy a LIN specifikáció az időzítésekre nagyon komoly hangsúlyt fektet a korábban már említett költséghatékonysági okokból eredő órajel eltérések miatt. Ezeket a névleges vagy nominális, illetve a maximális értékek megadásával teszi. Mind a teljes üzenetre (az elválasztó mező elejétől az ellenőrző összeg végéig eltelt időre: T_{keret}), mind pedig a fejléc ($T_{fejléc}$) illetve a válasz ($T_{válasz}$) mezők időbeli hosszára ad megkötéseket, a bitidő függvényében (T_{Bit}):

- A névleges, azaz nominális időzítések:
- $T_{fejléc_nom} = 34 * T_{Bit}$
- $T_{válasz_nom} = 10 * (N_{adat} + 1) * T_{Bit}$
- $T_{keret_nom} = T_{fejléc_nom} + T_{válasz_nom}$
- Maximális határ az időzítéseknél:
- $T_{fejléc_max} = 1,4 * T_{fejléc_nom}$
- $T_{válasz_max} = 1,4 * T_{válasz_nom}$
- $T_{keret_max} = T_{fejléc_max} + T_{válasz_max}$

4.1.6.2 LIN üzenetek típusai

A LIN 2.x szabvány alapvetően öt üzenettípust definiál:

- Az általános üzenetek (unconditional frame) a 0-tól 59-ig terjedő tartományt használhatják az azonosítókból. Egy adott azonosítóval rendelkező üzenetre csak egy csomópont válaszolhat, mely lehet egy szolga vagy a mester is. Ugyan válaszolni csak egy csomópont válaszolhat az adott azonosítóra, de venni több csomópont is veheti a LIN üzenetszórásos jellegéből adódóan. Tulajdonképpen az azonosító határozza meg, hogy az adott üzenetnek milyen adatot kell tartalmaznia. Ciklikus, azaz adott időközönként történő kommunikációra szolgál a LIN buszon levő eszközök között,

vagyis a mester periodikusan tud adatot lekérni a szolgáktól, illetve küldeni a szolgák felé.

- Az eseményvezérelt üzenetek (event triggered frame) az általános üzenetekhez hasonlóan a 0-tól 59-ig terjedő tartományt használhatják az azonosítókból. Ezek a LIN 2.x verziójú protokollban alkalmazott keretek elsősorban a ritkán előforduló események kezelésére szolgálnak. Alapvetően úgy működnek, hogy a mester a fejlécbe egy eseményvezérelt keret azonosítóját írja, melyre az érintett szolga eszközök csak akkor válaszolnak, ha új adat áll rendelkezésre (az utolsó lekérdezés óta megváltozott az érték). Az adat első bájtja mindig az adott szolga eszköz azonosítója (PID), így a mester tudja, hogy kitől jött a válasz, mivel az általános üzenetekkel ellentétben egy eseményvezérelt üzenetre több csomópont is válaszolhat. Előfordulhat ütközés, azaz hogy egyszerre több szolga is válaszolni akar az adott fejlécre, ekkor a mester feladata ezt lekezelni többszöri lekéréssel.
- A jelhordozó üzeneteknek azon csoportját, melynek tagjai ugyanazt az üzenethelyet használják, sporadikus (sporadic frame) üzeneteknek hívják. A sporadikus üzenetek lényege, hogy egy bizonyos fokú dinamikus viselkedést engednek meg egy determinisztikus viselkedésre tervezett hálózat, illetve ütemezés esetén úgy, hogy a látszólagos determinisztikusság fennmarad a tervezés szempontjából. Amikor a sporadikus üzenet küldése esedékes, az általános üzenetek frissítés igénylés szempontjából ellenőrzésen esnek át. Ha nincs frissítendő jel, akkor a sporadikus üzenetre szánt üzenethely üresen marad. Ha viszont van egy üzeneten belül egy, vagy több frissítendő jel, akkor ezen üzenet továbbítása meg fog történni. Ha több mint egy jel frissítése esedékes különböző üzeneteken belül, akkor a legmagasabb prioritású üzenet frissítése fog megtörténni. A fennmaradó alacsonyabb prioritású üzenetek nem fognak elveszni, hiszen az elkövetkező sporadikus üzenetekre szánt üzenethelyeknél a továbbításuk megtörténik. Egy frissítendő jeleket tartalmazó üzenet továbbítása addig tolódik, amíg nem ő lesz a legmagasabb prioritású.

  Általánosságban a többszörös sporadikus üzenetekhez ugyanaz az üzenethely van hozzárendelve, és ezen üzenetekből a legmagasabb prioritású kerülhet továbbításra az adott üzenethelyen. Ha egyetlen általános üzenet továbbítása sincs függőben, akkor az üzenethely üresen marad (a csomópontok adásmentesen figyelik a buszt).

- A diagnosztikai keretek (diagnostic frame) mindig a 60-as illetve 61-es azonosítóval rendelkeznek. Ezek az adott LIN klaszterhez tartozó szolga egységek diagnosztikájához és konfigurálásához használt üzenetek, melyek adatmezője mindig 8 bájt hosszú. A 60-as azonosítóval rendelkező keretek a mester kérései a szolga felé. Ezen keretek első adatbájtja többnyire a szolga eszköz azonosítóját tartalmazza, és többek között alkalmazható például arra, hogy felszólítsa a szolga eszközt, hogy aludjon el. A kérésre a szolga eszköz általában egy 61-es azonosítóval rendelkező üzenettel válaszol.
- Léteznek még fenntartott keretek (reserved frame), melyek a 62-es és 63-as azonosítókat hivatottak későbbi felhasználásra fenntartani (LIN 2.x protokollt alkalmazó rendszereknél alapértelmezetten nem használják).

4.1.7 MOST

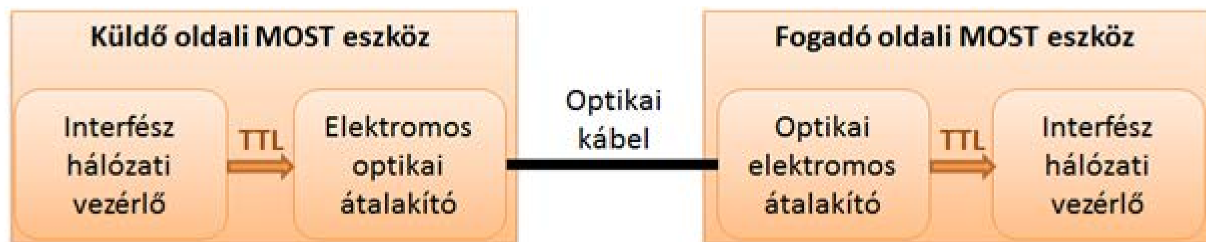
A MOST egy speciálisan a gépjárművek számára kifejlesztett kommunikációs technológia, a multimédiás adatok továbbítására. Magának a MOST névnek a jelentése (Media Oriented System Transport) is erre utal. Két fő követelmény fogalmazódott meg a gyártók részéről a MOST-al szemben, amelyeket hiánytalanul teljesít is:

- A MOST buszon video-, és audio jelek, navigációs adatok, és más szolgáltatások mellett vezérlő jelek is átvihetőek legyenek
- Maga a MOST technológia logikai felépítése alkalmas legyen arra, hogy a járműben található sokféle, és komplex adatokat egyszerre kezelje. Ezen kívül a teljes rendszernek a funkcióit és feladatait is rendszerezi azáltal, hogy az alapjaitól fogva úgynevezett függvényblokkokból (Function Block) épül fel.

4.1.7.1 MOST fizikai rétege

A jelek továbbítását nagy sebességű illetve sávszélességű MOST hálózatok esetén, az esetek többségében műanyag optikai szállal valósítják meg ezzel biztosítva azt, hogy a működés közbeni elektromágneses interferencia ne jöjjön létre. Ahhoz, hogy az adatok továbbíthatóak legyenek optikai kábelen keresztül, egy 1mm átmérőjű műanyag optikai szálra és vörös hullámhossz-tartományba eső fényt kibocsátó LED-re van szükség.

A MOST protokoll megvalósításához azért alkalmaznak optikai kábelt, mert rendkívül gyors adatátvitelt tesz lehetővé, fizikailag könnyebb és rugalmasabb, mint más kábelszabványok, továbbá megfelel a szigorú elektromágneses kompatibilitási követelményeknek. A MOST busz feladata, hogy a küldő oldali MOST eszköz által létrehozott elektromos jelből konvertált optikai jelet továbbítsa a fogadó oldali MOST eszköz optikai-elektromos átalakító egysége felé (4.18. ábra).



4.18. ábra A MOST buszrendszer működése

4.1.7.2 MOST szabványok

A MOST hálózatokra csatlakoztatott eszközök egy folytonos alapjelhez szinkronizálódnak. Nincs szükség puffer tárolók használatára, sem mintavételezett konverzióra, így a hálózat hardverfelépítése meglehetősen költséghatékony.

A szinkronizációs órajelre azért van szükség, hogy a különböző adatfolyam-átviteli csatornák és a vezérlőjel csatorna ütemezetten tudjon működni. A vezérlőjel csatorna feladata, hogy megállapítsa, hogy melyik csomópont melyik átviteli csatornát használja. Amint a kapcsolat kiépült, azonnal megindulhat a felek közötti kommunikáció anélkül, hogy újabb címzésre vagy csomagcímkézésre lenne szükség.

A kommunikációban részt vevő adatfolyam egy előre lefoglalt sávszélességgel rendelkezik, amelyből következik, hogy nincs megszakításkezelés, sem adatütközés, és az átviteli sebesség állandónak tekinthető.

A szabvány definiálja a MOST hálózat fizikai jellemzőt, így megadva a kommunikációs csatorna (kábel) típusát, továbbá a kommunikációs protokollt és a szoftveres keretrendszert. A fizikai réteg megvalósítható műanyag optikai szálas (POF), üvegszálas (PCS), árnyékolt csavart érpárral (STP), árnyékolatlan csavart érpárral (UTP) és koaxiális (COAX) kábellel egyaránt. A kábel kiválasztásával befolyásolható az elérhető maximális adatküldési sebesség, valamint a sávszélesség is korlátozódhat.

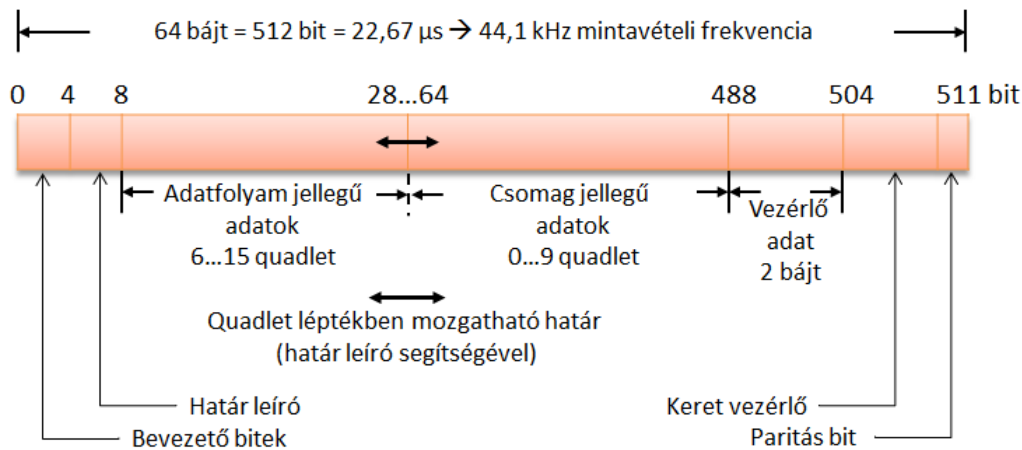
A MOST hálózat akár 64 különböző eszköz egyidejű kezelését is lehetővé teszi csillag topológiában. A csillag topológia sérülékeny volta miatt biztonságkritikus esetekben a meghibásodás elkerülésének érdekében applikáció-redundáns duplagyűrűs konfigurációt építenek ki. A gyűrű topológiára jellemző, hogy ha a gyűrű bármely részén meghibásodás lép fel, akkor az adatátvitel leáll, ezért célszerű alkalmazni az előzőekben említett duplagyűrűs konfigurációt.

Mivel a MOST kommunikációs protokoll elsősorban a vezető és az utasok kényelméért felelős eszközök vezérlésére szolgál, ezért a hálózati eszközök csatlakoztatásának egyszerűnek kell lennie, amelynek megvalósítására felhasználják a Plug&Play alapú technológiát, melynek során a saját eszközök automatikusan inicializálásra kerülnek. A MOST fejlődése során több MOST szabvány is piacra került:

- A legelső szabvány a MOST 25 volt. Ezen szabvány körülbelül 23 MBaudos sávszélességet kínál szinkron és csomag alapú, optikai szálon történő adatátvitelhez. 60 fizikai csatornát definiál, amelyből a felhasználó 4 bájtos csoportokat alakíthat ki. Támogat 15 tömörítetlen sztereó hangcsatornát CD minőségben, vagy akár 15 MPEG-1 csatornát audio-video átvitelhez, melyek mindegyike 4-4 fizikai csatornát igényel. Továbbá a MOST 25 biztosítja az adatfolyam felügyelésére alkalmas vezérjel csatornát, amely egyben a referencia adatok küldését is elvégzi. A vezérjel üzenetek konfigurálják a MOST eszközöket, tehát irányítják a szinkron-aszinkron adatátvitelt.

  A MOST 25 protokollban alkalmazott keretek hossza 64 bittől egészen 512 bitig terjedhet. Az adatátvitel legkisebb részét (2 bájt) a vezérlőjel üzenetek adják, valamint 32 bájt van fenntartva az adminisztrációs hálózatok és eszközök számára. Az első és utolsó bájt ellenőrző információkat tartalmaz.

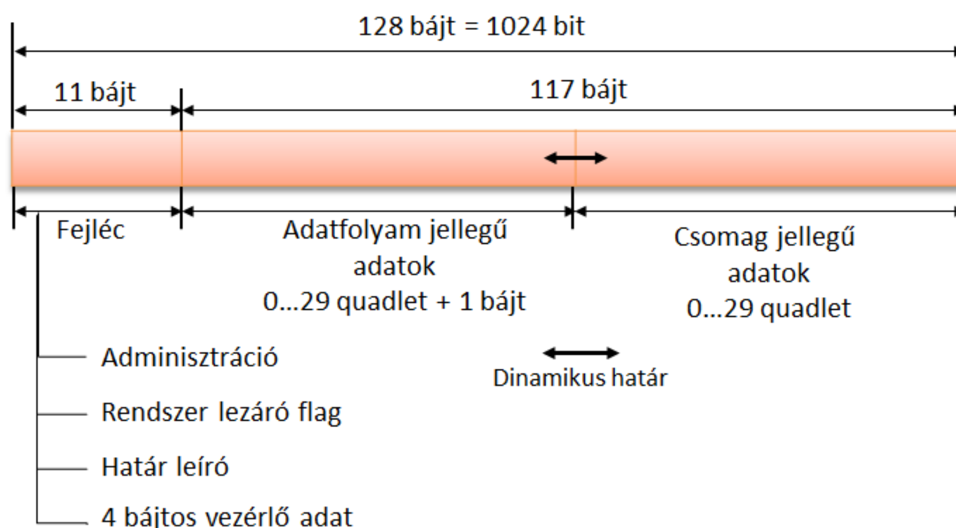
  Az adatfolyam-átviteli és a csomagkapcsolt adatátviteli csatorna quadletekre bontva osztozik a sávszélességen. A quadlet szó jelentése: 4 bájtos szóhossz. Az adatfolyam csatorna esetén a hálózati interfész-vezérlőhöz tartozó időzítés 6-15 quadletes felosztásban van konfigurálva, míg az aszinkron csomagkapcsolat adatátvitel esetén ez jóval kevesebb (0-36 bájt). A quadletek felosztásának megváltoztatásával a mesternek azonnali újraszinkronizálást kell végrehajtania. A protokollban alkalmazott mintavételi frekvencia 44,1 kHz (4.19. ábra).



4.19. ábra A MOST 25 esetében alkalmazott átviteli protokoll

- A MOST 50 megduplázza a MOST 25 által biztosított sávszélességet, valamint támogatja az 1024 bites kerethosszúságot. Fizikai réteg tekintetében lehetővé teszi az elektromos és az optikai megoldásokat egyaránt. A MOST 50 elsősorban optikai és elektromos fizikai rétegek támogatására szolgál, de léteznek a MOST 50 Intelligens hálózati interfész vezérlők (INIC), amelyek csak UTP kábelen keresztül támogatják az adatátvitelt.

  Ugyan a MOST 50 megnövelt sávszélességet biztosít, bár ennek ellenére az audio jelfolyam mintavételi frekvencia nem változott meg (44,1 kHz), de a keretek hossza növelhető 1024 bitre. Ezen protokollban a 4 bájtos vezérlőjel adat már bekerült a 11 bájtos fejlécbe, ahol többek között megtalálhatóak a lezárási flagek és a quadlet felosztási adatok is. A rendelkezésre álló sávszélesség dinamikusan igazítható az igények szerint, hiszen az adatfolyamok és csomagkapcsolt adatátvitelek egyenként maximum 29 quadletből állhatnak (4.20. ábra).



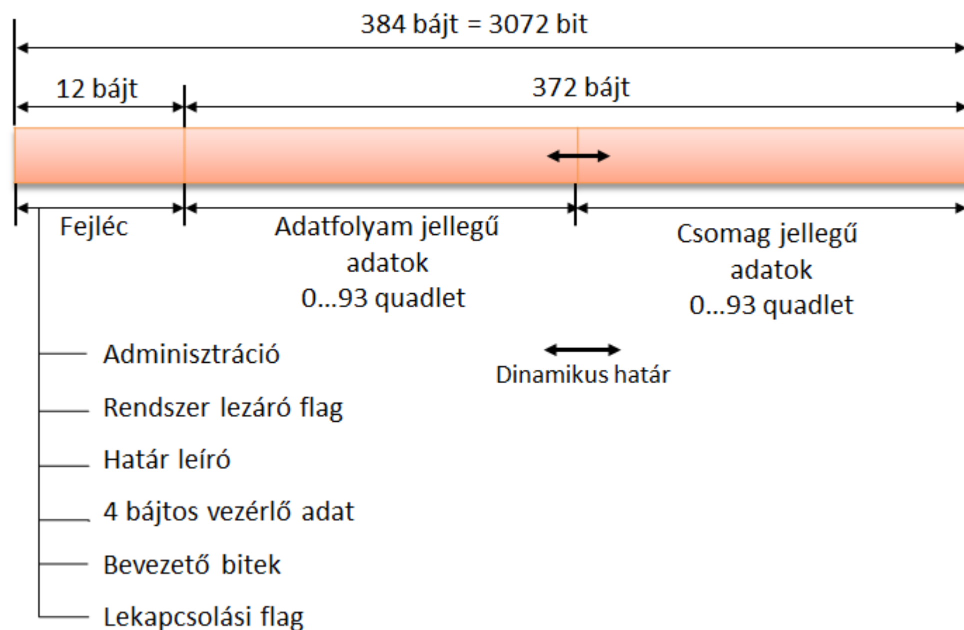
4.20. ábra A MOST 50 esetében alkalmazott átviteli protokoll

- A jelenleg legfejlettebbnek tartott MOST szabvány a MOST 150, amelynél a keret maximális hossza 3072 bit értékűre növekedett, így lehetővé téve a közel 6-szoros sávszélességet a legelső szabványhoz képest.

  A MOST 150 magában foglal egy Ethernet csatornát (állítható sávszélességgel), az eredetileg kialakított csatornák mellett. A fejlett funkciói és megnövelt sávszélessége lehetővé teszi multiplex hálózati infrastruktúrák kialakítását, így a legmodernebb információs- és szórakoztató rendszerek probléma nélkül beépíthetőek.

  Ezen protokoll esetén szintén 44,1 kHz-es mintavételi frekvencia alkalmazott, de már háromszoros sebességgel képes működni. A keretek hossza 3072 bit, amelyből 12 bájt a fejléc, a maradék 372 bájt pedig dinamikusan oszlik el az adatfolyamok és csomagkapcsolt adatátvitelek között. A szinkron adatfolyamok alkalmasak arra, hogy multimédiás adatokat valós időben továbbítsanak (4.21. ábra).

  Az adatokat ciklikusan továbbítjuk a hálózaton keresztül. Amennyiben kommunikációs hiba került észlelésre a hálózaton, akkor az adat továbbítása megszakad.



4.21. ábra A MOST 150 esetében alkalmazott átviteli protokoll

4.1.8 FlexRay

A FlexRay nem titkolt kezdeti célja a CAN bizonyos mértékű kiváltása, valamint az ún. x-by-wire „vezeték általi" (elektromos vezérlés, hagyományos mechanikus helyett) technológiák alkalmazási lehetőségeinek megteremtése volt. Már itt fontos megjegyezni, hogy a FlexRay protokoll viszonylag nagyteljesítményű és bonyolult felépítésű vezérlőt igényel. Ennek köszönhetően lassú az elterjedése, és többnyire ott alkalmazzák, ahol a CAN teljesítményét tekintve nem megfelelő.

A FlexRay esetében támogatott mind a nagy sebességű szinkron, mind az aszinkron átviteli mód is, melyekkel akár 10Mb/s-os vagy még magasabb sávszélesség is elérhető. A FlexRay-

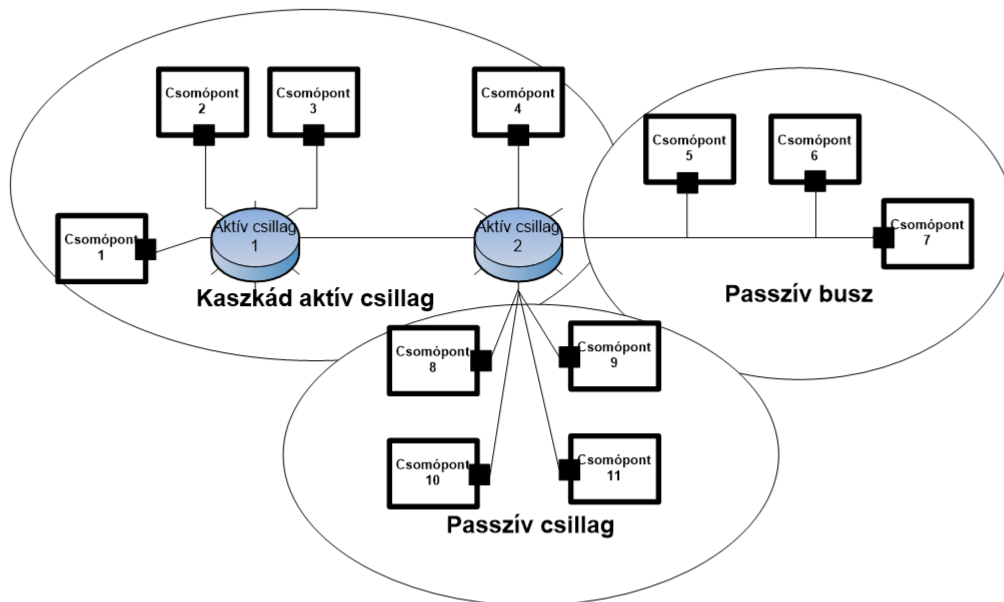
nek több verziója is elérhető, melyek közül a legelterjedtebb a 2.1-es és az afeletti verziók, melyek főbb jellemzői, illetve eltérései a korábbi verzióktól:

- 2x10Mb/s sávszélesség a két, egyenként 10Mb/s sávszélességgel rendelkező kommunikációs csatorna támogatásával, így akár hússzor nagyobb sávszélesség is elérhető, mint a CAN rendszer esetében.
- Idő szinkronizáltság, azaz az adathozzáférést idő szinkronizációhoz köti, melyet a protokoll végez automatikusan, hogy elérhetővé váljon az adat az alkalmazások részére. Az időalap (macrotick) pontossága 0.5-10 μs között van, általában 1-2 μs.
- Ismert üzenetkésleltetés garantált szórással, vagyis a kommunikáció periodikusan ismétlődő körökből áll. Minden körben van egy speciális fix helyen lévő üzenet, melyből a vevő tudja mikor érkezett meg az adat, ezáltal a beérkezési idő igen jól becsülhető, és garantált a kis szórás.
- Redundáns és nem redundáns kommunikáció támogatása. A FlexRay redundánsan továbbítja az egyes üzeneteket, hogy további rétegek számára is biztosítsa a hálózat megbízhatóságát. A programozható átviteli redundancia megengedi a tervezőnek, hogy redundáns továbbítást használjon, a lehető legjobb sávszélesség kihasználás érdekében.
- Tervezéskor a fő hangsúly a rugalmasságra összpontosult. Szabadon választható meg, hogy redundánsan vagy nem redundánsan továbbítódnak az üzenetek. A rendszer optimalizálható használhatóságra, vagy teljesítményre mindezt úgy, hogy a rendszer kiterjeszthető a csomópont (node) beállításainak változtatása nélkül. Továbbá különböző busztopológiákat is támogat (busz, csillag…), valamint változtathatóak a beállítási paraméterei (üzenethossz, kommunikációs ciklus hossza, stb.), így beállítható a kommunikációs rendszer, hogy megfeleljen az alkalmazás követelményeinek.

A FlexRay arra lett tervezve, hogy kiszolgálja az új technológiákat és alkalmazásokat, de köszönhetően a nagy sávszélességnek és hálózati rugalmasságnak, teljesíti több jelenlegi, autoiparban használt alkalmazás szükségleteit is:

- A CAN felváltása azokban az alkalmazásokban, ahol nagyobb sávszélességre van szükség, mint amit a CAN biztosítani tud, vagy ahol kettőnél több CAN buszt használnak párhuzamosan.
- A nagy sávszélességnek köszönhetően, alkalmas az autók gerinchálózatának kialakításhoz, biztosítva a kapcsolatot a sok, különálló független hálózat között.
- Valós idejű alkalmazásokra, és elosztott rendszerirányításra is alkalmas. A garantált időtartamú kommunikációs ciklusoknak és alacsony szórásnak köszönhetően, a FlexRay rendszer megfelel az elosztott rendszerek szigorú, valós idejű követelményeinek.
- A FlexRay egymaga nem teszi a rendszert biztonságossá, de a variációk sokfélesége, amit a rendszer nyújt, teszi lehetővé a FlexRay alapú biztonság-orientált rendszerek fejlesztését, mint pl. a x-by-wire rendszerek.

Sok különböző módja van, hogy kialakítsunk egy FlexRay clustert (csomópontokból álló busz vagy csillag topológiájú kommunikációs rendszert). Lehet egy-, illetve kétcsatornás busz, csillag elrendezésű vagy hibrid megoldásokat is tartalmazó rendszer (4.22. ábra).

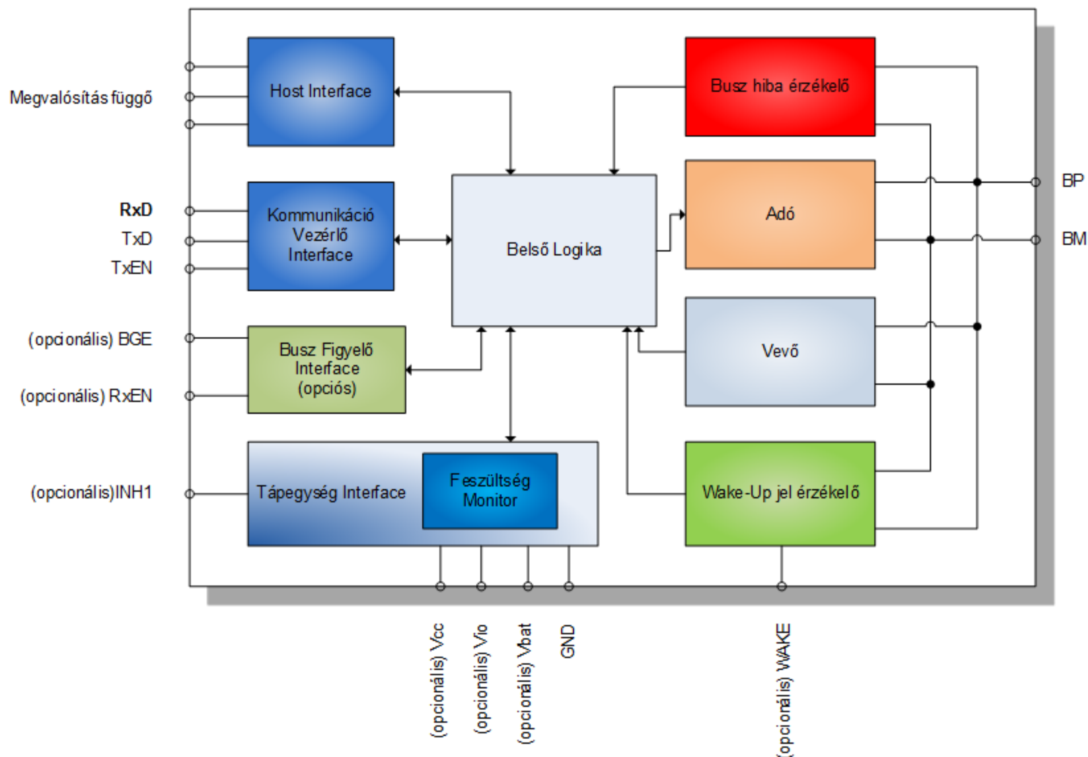


4.22. ábra Egy FlexRay hibrid topológiájú hálózat felépítése

A protokoll rétegeinek szempontjából a FlexRay három réteget definiál: az alkalmazási, adatkapcsolati és fizikai réteget. Az átvitel során tipikusan minden egyes csatorna egy csavart érpárból áll, ahol a CAN-hez hasonlóan differenciális feszültségmérésen alapul a bitek átvitele.

4.1.8.1 Busz meghajtó (driver)

Az elektronikus busz meghajtó (BD) realizálja a fizikai kapcsolatot a FlexRay csomópont (node) és a csatorna között. A meghajtó biztosítja a küldést és fogadást a buszon a csomópont modulnak, kétirányú időmultiplexelt bináris adatfolyam továbbításához. A továbbítás és fogadás mellett a meghajtó szolgáltatja az eszközt az alacsony energiaszintű működéshez, a tápfeszültség figyeléséhez valamint a buszhiba detektálásához, továbbá védelmet nyújt az elektronikus vezérlőegység és a busz közötti elektromos kisülésekkel szemben.



4.23. ábra A busz meghajtó belső logikai felépítése

A busz meghajtónak két fő működési módja van: a BD_Normal és BD_Standby, amelyeket kötelező implementálni. A fentieken kívül két további opcionális állapotot vehet fel a busz meghajtó, még pedig a BD_Sleep és a BD_Receive_Only módokat. Ugyanakkor ezek még tovább bővülhetnek egyéb termék-specifikus módokkal.

- A BD_Normal módban a meghajtó normál módban fogadhat vagy küldhet adatfolyamot a buszon.
- A BD_Standby mód egy alacsony energia felvételű készenléti állapot, melyben a meghajtó nem képes adatot küldeni és fogadni a buszról, de detektálni tudja az úgynevezett Wake-Up (ébresztési) eseményeket.
- A BD_Sleep mód megegyezik az előbb leírt készenléti móddal, de itt a meghajtó kimenetén Sleep jelzés jelenik meg.
- A BD_Receive_Only állapotban csak fogadni tudunk adatot, továbbítani nem.

4.1.8.2 A protokoll irányítása

A protokoll alapvető viselkedését négy fő mechanizmus szabályozza:

- Kódolás és dekódolás (Coding and Decoding): Ha két csatornával rendelkezik minden csomópont, akkor szükség lesz két független halmazra a kódoló/dekódoló folyamatokhoz az A és B csatorna számára. A kódolás/dekódolás három folyamatból épül fel: egy fő kódolásból illetve dekódolásból (CODEC), és két alfolyamatból: bitválasztó folyamatból (BITSTRB), valamint felébresztési (Wake-Up) minta dekódolásból (WUPDEC). A bitválasztó folyamat dönti el, hogy mely csatornára kell továbbítani az adott jelsorozatot. A felébredési minta dekódolás feladata a speciális ébredési szimbólumok érzékelése, és a csomópont felébresztésének a megkezdése.

- Közegelérés (Media Access Control): A FlexRay időosztásos multiplexelést használ a közegeléréséhez, ezzel megakadályozva a keretek ütközését. Ugyanakkor eseményvezéreltnek is kell lennie az igények szerint, amit úgy érnek el, hogy apróbb logikai egységekre bontják a rendelkezésre álló időt (statikus időosztásos mód (statikus szegmens), dinamikus mód (dinamikus szegmens), mely az eseményvezérelt rész, valamint a szimbólum ablak a hálózat vezérléshez), amely egységeknek viszont továbbra is az időosztásos multiplexelés elvét kell követniük. Így egy jól becsülhető késleltetésű rendszer kapható.
- Keret és szimbólum feldolgozás (Frame and Symbol Processing)
- Órajel szinkronizálás (Clock Synchronization)

Ezeket összefogva, a Control Host Interface (CHI) biztosítja a lehetőséget, hogy megszerkesztett formában férjen hozzá a csomópont a 4 fő protokoll mechanizmushoz, beleértve a protokollirányítást (POC, protocol control), ami visszacsatolást biztosít a csomópont (host) felé. A különböző mechanizmusokból néhány a későbbiek során részletesen is ismertetésre kerül.

4.1.8.3 Hibakezelés

A POC kétféleképpen reagálhat egy hibára. Súlyos/lényeges hiba esetén a POC egyből felfüggesztett (halt) állapotba kerül:

- Termék specifikus hibák
- Freeze parancs hatására hibához vezető feltételeket detektál a csomópont
- Végzetes hibához vezető feltételek kerültek detektálásra a POC, vagy a fő folyamatban

A CAN-hez valamelyest hasonlóan a POC is tartalmaz egy 3 állapotú hibatűrő degradációs modellt, amely elvisel adott számú hibát egy időperiódusban. Ebben az esetben a POC nem kerül egyből felfüggesztett állapotba. Ez a modell annak érdekében lett kitalálva, hogy reagáljon bizonyos hibafeltételekre, amelyeket az óraszinkronizációs folyamat során érzékel, de nem igényel azonnali beavatkozást. Így hibatűrést lehet biztosítani a szinkronizáció során. Ezzel el lehet kerülni az azonnali felfüggesztést, így lehetőség nyílik a hiba nagyságának és természetének becslésére. Magát a modellt a következő három POC állapot alkotja:

- Normál aktív állapotban a csomópont hibamentesen, vagy egy minimális hibahatáron belül működik. Ekkor megengedhető, hogy normál tartományban maradjon a POC. Vagyis ha a csomópont megfelelően van szinkronizálva egy kommunikációs rendszerhez, akkor folytathatja az átvitelt anélkül, hogy a többi csomópont átvitelét megszakítaná.
- A normál passzív állapot esetén feltételezzük, hogy a szinkronizáció a klaszter többi részéhez képest alacsony. Ekkor a keret továbbítása nem folytatható, mivel az ütközéseket okozhatna a többi csomópont kereteinek átvitelében. A keretek fogadását tovább folytathatja a csomópont ebben az állapotban, hogy a csomópont elvégezze az újraszinkronizálást és ebből az átmeneti állapotból újra normál aktív állapotba kerüljön.

- A felfüggesztett állapotba akkor lehet átlépni, ha további hibák kerültek detektálásra a passzív állapotban, vagy sok hiba fordult már elő. Ebből az állapotból már nem lehet visszakerülni egyből a normál aktív tartományba, ugyanis a POC leállítja a fő mechanizmusokat, hogy előkészítse és újrainicializálja a csomópontot.

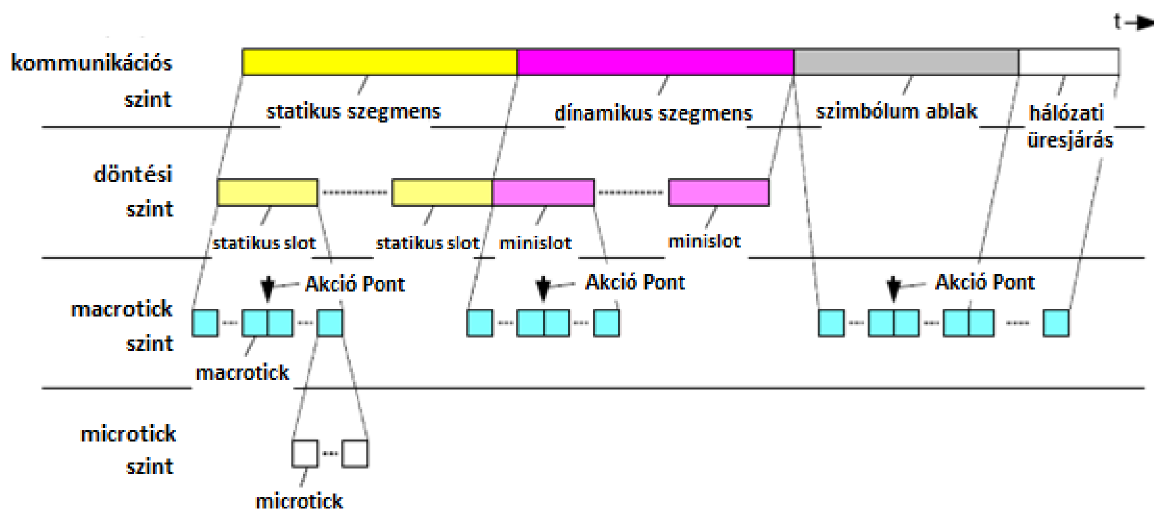
4.1.8.4 Keretezés és keret felépítés

A FlexRay protokoll egyszerre idő- és eseményvezérelt. Ezt úgy lehet elérni, hogy a kommunikáció ciklusokra van osztva (4.24. ábra), melyben van egy fix időosztásos szegmens (statikus szegmens) (4.25. ábra), egy dinamikusan allokálható időosztásos rekeszeket (slot) tartalmazó szegmens (dinamikus szegmens) (4.26. ábra) és egy szimbólum ablak (4.27. ábra), mely különböző hálózat irányítási feladatokat lát el. Ezenkívül a ki nem használt időt a kommunikációs ciklusban hálózati üres járásnak (NIT-nek) nevezik.

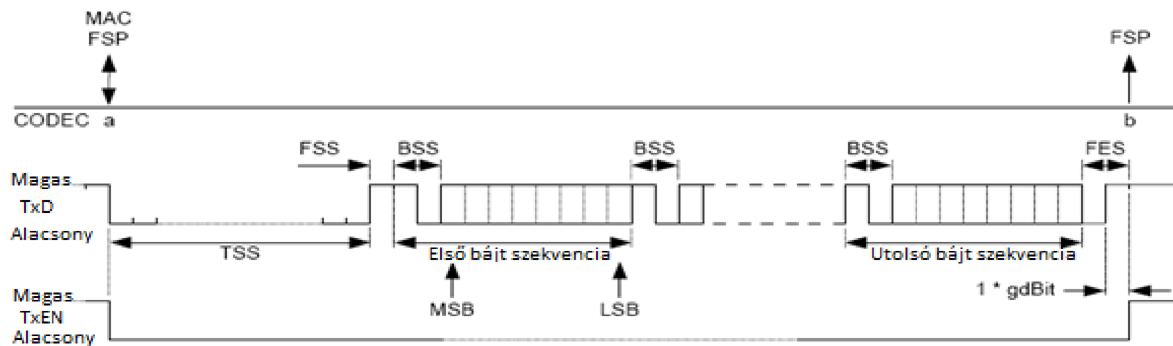
A statikus és a dinamikus szegmens valamint még a szimbólumablak is tovább bontható kisebb logikai egységekre. A statikus szegmens statikus rekeszekre (slot-okra) bontható, melyek meghatározott ütemhosszúságú logikai egységek, melyek előre meghatározott üzenetet tartalmazhatnak.

Dinamikus esetben minislotokról lehet beszélni, melyek meghatározott ütemhosszúságú logikai egységek. A statikus szegmens statikus rekeszeivel ellentétben nem előre rögzített, hogy melyik csomópont melyik rekeszt, milyen üzenet továbbítására használja, hanem igény szerinti dinamikus kiosztásúak is lehetnek.

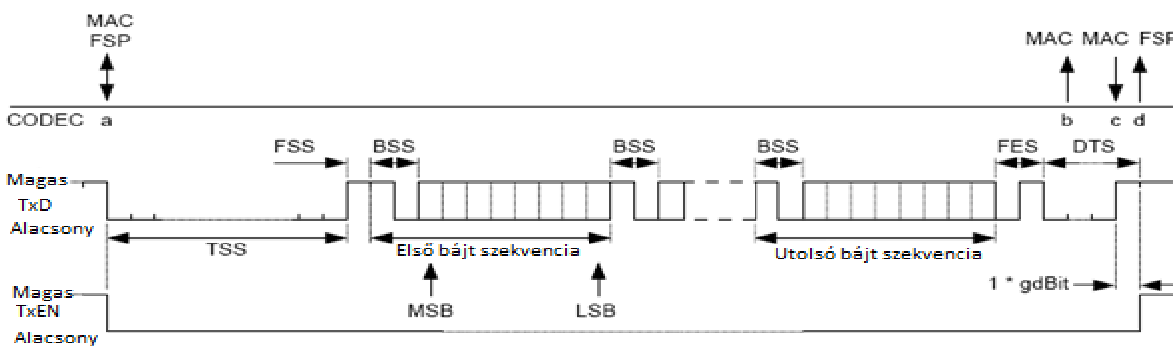
A szimbólumablakon belül egyedülálló szimbólumot lehet küldeni. Választást különböző küldők között nem biztosít a protokoll a szimbólumablak számára, ha mégis szüksége lenne erre, akkor azt egy magasabb szintű protokoll fogja biztosítani számára.



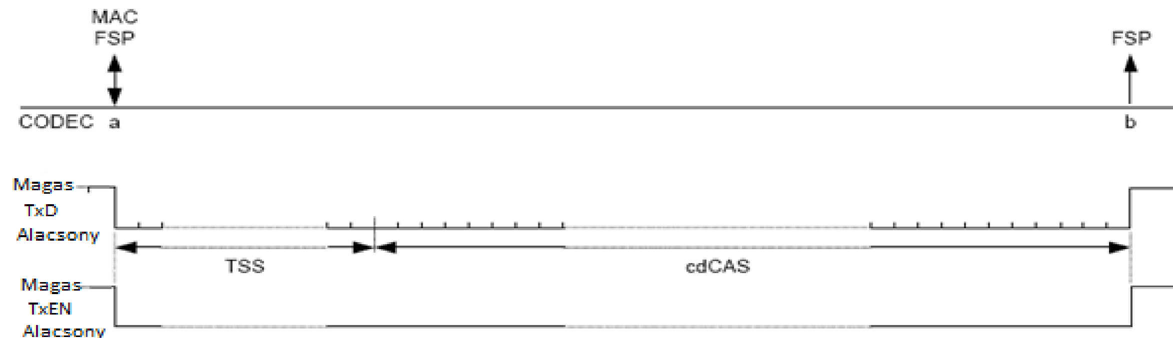
4.24. ábra FlexRay kommunikációs ciklus szerkezete



4.25. ábra: Egy statikus szegmens felépítése



4.26. ábra: Egy dinamikus szegmens felépítése



4.27. ábra: Egy szimbólum ablak felépítése

A keretek küldése több különböző szekvenciára bontható szét:

- Az átvitel kezdetét jelzi a TSS, azaz a Transmission Start Sequence**.** Arra szolgál, hogy kiépítse a megfelelő kapcsolatot a hálózaton keresztül. Egy küldő csomópont generál egy TSS-t, ami folytonos alacsony szintet generál meghatározott ideig.
- A TSS követi a keret kezdetét jelző FSS-t, azaz a Frame Start Sequence, melynek a feladata, hogy kompenzálja a lehetséges kvantálási hibát az első BSS-ben az FSS után. Az FSS egy magas értéket fog tartalmazni meghatározott, névleges bit ideig. A csomópont hozzáfűzi az FSS-t a bitfolyamhoz, közvetlenül a küldendő keret TSS-e után.
- A bájtok kezdetét jelöli a BSS, azaz a Byte Start Sequence, mely biztosítja az időzítési információkat a fogadó eszközöknek. A BSS tartalmaz egymás után egy magas és egy alacsony bitet, mindegyiket egyaránt a névleges bitidő idejéig. A keretben található

adat minden bájtja egy kiterjesztett sorozatként továbbítódik, ami tartalmazza a BSS-t és az azt követő 8 adat bitet.

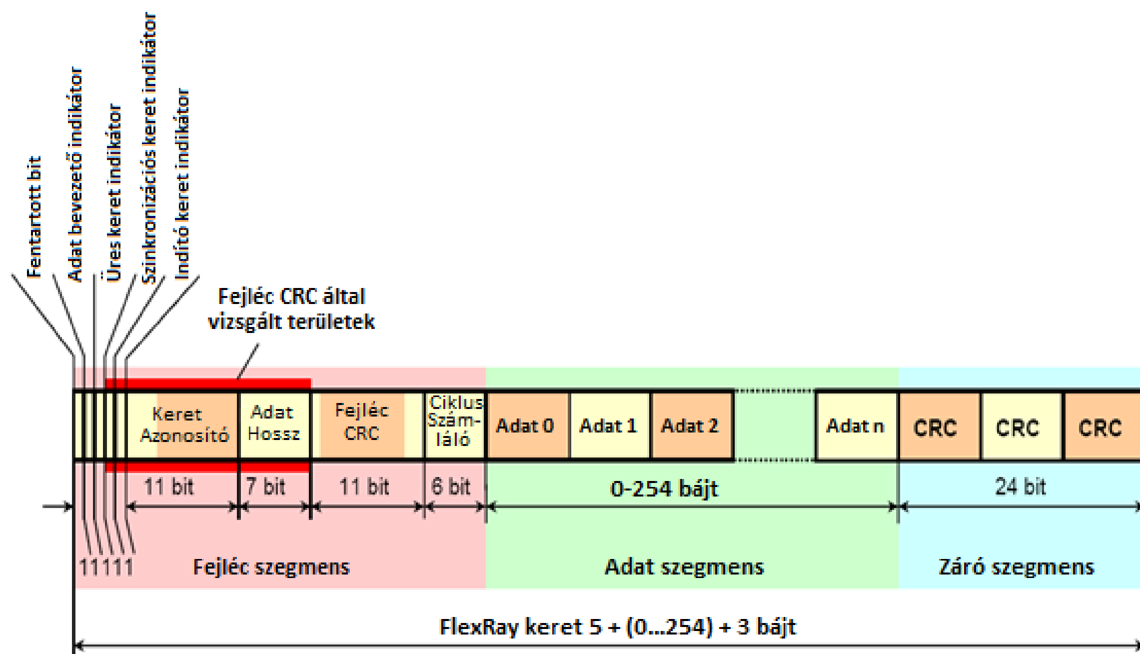
- A FES, azaz Frame End Sequence jelöli meg az utolsó bájtsorozatot az adott keretben. Az FES tartalmaz egymás után egy alacsony és egy magas bitet, melyeket névleges bitideig tart fenn. A csomópont a lezárás jelzésére beszúr egy FES-t a bitfolyamba, közvetlenül a keret utolsó kiterjesztett bájtsorozata után.
- Dinamikus keretek továbbítása esetén létezik csak a dinamikus nyomkövetésre szolgáló szekvencia, a DTS (Dynamic Trailing Sequence). A feladata, hogy jelezze a minislot akciópont (AP) (az akciópont egy pillanat az időben, melyben a csomópont végrehajt egy speciális akciót, hogy beállítsa a saját lokális időegységét) pontos helyét az időben, és megelőzze, hogy a csatornán a fogadó idő előtt detektáljon üresjáratot. Mikor dinamikus szegmensben történik egy keret elküldése, akkor a csomópont a DTS-t közvetlenül a keret FES sorozata után küldi el. A DTS két részből áll: egy változó hosszúságú periódus a TxD (a TxD - Transmit Data - jel feladata, hogy továbbítsa az aktuális jelsorozatot a buszmeghajtó felé, mely a kommunikációs csatornára helyezi a továbbítandó adatot) kimenet alacsony szintjéhez, ezt követi egy fix hosszúságú periódus a TxD kimenet magas szintjéhez. Az alacsony szintű periódus minimális hossza egy nominális bitidőnyi időtartam. Ezután a minimális hossz után a csomópont a TxD kimenetet alacsony szinten hagyja a következő minislot akciópontig, ahol is a csomópont a TxD kimenetet magas szintre állítja és egy névleges bitidőnyi késleltetés után a TxEN (a TxEN - Transmit Data Enable Not - jelzi a buszmeghajtó felé, hogy a megfelelő csatorna TxD vonalát engedélyezze, hogy adatot küldhessen rajta) kimenetet is magas szintre állítja. A DTS hossza változó: két névleges bitidőnyi időtartamtól egészen két névleges bitidőnyi időtartamig, egy minislot időtartamáig terjedhet.

Maga a FlexRay keret több részből épül fel (4.28. ábra):

- A fenntartott (Reserved) bit egy 1 bites mező, mely a jövőbeni protokoll bővítésére van fenntartva. Ha az alkalmazás nem használja, akkor a küldő csomópont 0-ra állítja és a fogadó csomópont pedig figyelmen kívül hagyja a fenntartott bitet.
- A fenntartott bitet az adatbevezető indikátor (Payload preamble indicator) követi, mely szintén 1 bit hosszúságú. Azt jelzi, hogy a továbbítandó keret tartalmaz-e egy opcionális vektort az adatszegmensben.
- Az üres keret indikátor (Null frame indicator) 1 biten jelzi, hogy a továbbítandó keretünk tartalmaz-e hasznos adatot, vagy sem. Egy üres keretet fogadó csomópont mégis információt kaphat a keretet illetően.
- Szinkronizációs keret indikátor (Sync frame indicator) szintén egy 1 bites jelző flag, melyet a szinkronizációs folyamatokhoz használ a protokoll.
- Az indító keret indikátor (Startup frame indicator) 1 biten megadja, hogy indító keret érkezett-e vagy sem. Ezeknek a kereteknek speciális feladatuk van az indító mechanizmus folyamán.
- A keretazonosító (Frame ID) egy 11 bites mező, mely definiálja, hogy az adott keret melyik slot-ba kerül továbbításra. Minden egyes keretazonosító csak egyszer

használható egy csatornán egy kommunikációs ciklus alatt, valamint minden egyes keret amely továbbítódik, kell hogy kapjon egy azonosító számot.

- Az adathossz (Payload length) mező 7 biten adja meg az adatszegmens hosszát. A benne kódolt információ az adatszegmens hosszának a felét adja meg bájtban, de nem tartalmazza a fejléc, illetve a záró szegmens hosszát.
- A fejlécre vonatkozóan van egy 11 bites ellenőrző összeg, azaz CRC mező, mely a kiszámításakor figyelembe veszi a szinkronizációs keret és indító keret indikátor, keretazonosító és adathossz mezőket.
- A ciklusszámláló (Cycle count) 6 biten jelzi a küldő csomópont számára a ciklusszámláló értékét (0 – 63) a küldés pillanatában. A továbbítása pedig a legmagasabb helyi értékkel kezdődik.
- Az adatszegmensben (Payload segment) továbbítódik a tényleges információ, mely 0 – 127 db kétbájtos szót tartalmaz. Ezért mindig páros számúnak kell lennie a továbbított bájtoknak, mivel a fejrészben tárolt Adathossz mező is a továbbított adat hosszának felét tárolja.
- A zárószegmens (Trailer segment) vagy hibaellenőrző tag egy 24 bites CRC ellenőrző összeget tartalmaz az egész keret számára, amit a fejléc és adat szegmensekből számol, figyelembe véve minden értéküket.



4.28. ábra: FlexRay keret

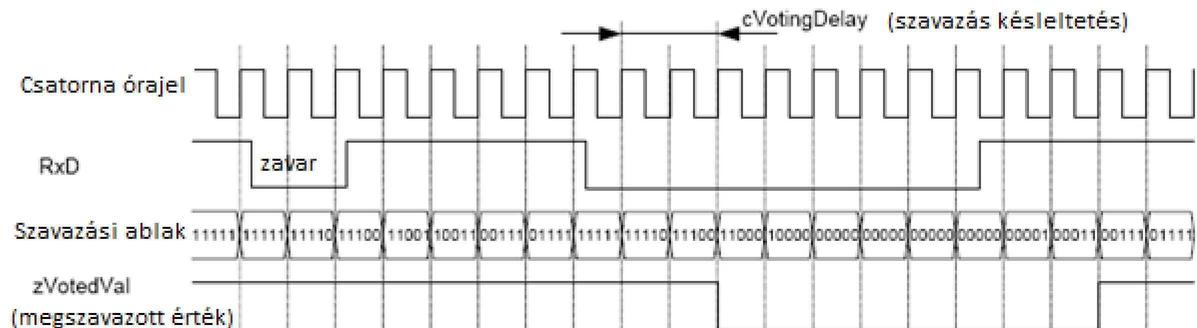
4.1.8.5 Mintavételezés és többségi szavazás (Majority voting)

A fizikai jellegű hibák kiküszöbölésére szolgál a többségi szavazás, többszöri mintavételezés segítségével. A csomópont mintavételezi az RxD (a buszmeghajtó használja az RxD - Receive Data - jelet, hogy továbbítsa az aktuálisan fogadott adatokat) bemenetet úgy, hogy az egyes mintavételezési periódusokban a csomópont mintát vesz és tárolja az RxD bemenet értékét.

A csomópont a mintavételezett RxD jelen hajtja végre a többségi szavazást. Az eljárás célja, hogy szűrje a bemeneti jelet, azaz a zajokból származó téves értékek számát csökkentse. Itt a

zaj alatt olyan esemény értendő, ami megváltoztatja az aktuális állapotát a fizikai rétegnek oly módon, hogy az észlelt logikai állapotot átmenetileg megváltozatta egy érték, ami különbözött a küldöttől.

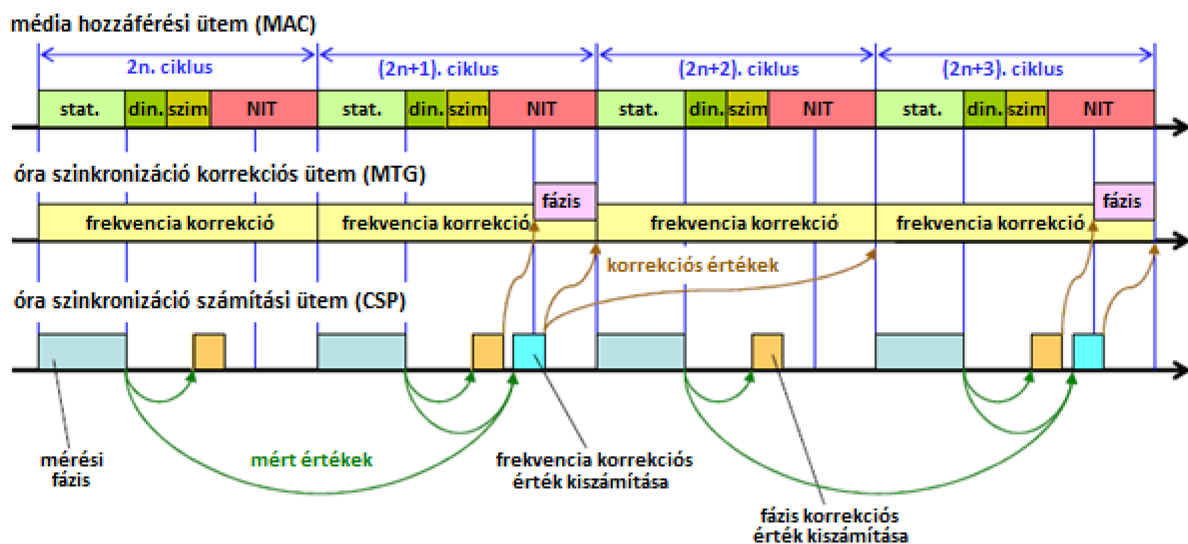
A dekóder folyamatosan értékeli az utolsó eltárolt mintát, és számolja a magas szintű minták számát. Ha a minták többsége magas szintű volt, akkor az ún. szavazó egység kimenet magas szintet fog adni, egyébként alacsony szintet küld (4.29. ábra).



4.29. ábra: A FlexRay többségi szavazás algoritmusának a szemléltetése

4.1.8.6 Óra szinkronizáció

Mivel a FlexRay protokoll időosztásra épít és nem központi időt használ, így fontos, hogy minden csomópont (host) szinkronizálja a belső óráját. Ezt a feladatot két párhuzamosan futó folyamat, a macrotick ütemgeneráló (Macro Tick Generator = MTG), és az óra szinkronizációs folyamat (Clock Synchronization Process = CSP) látja el (4.30. ábra).

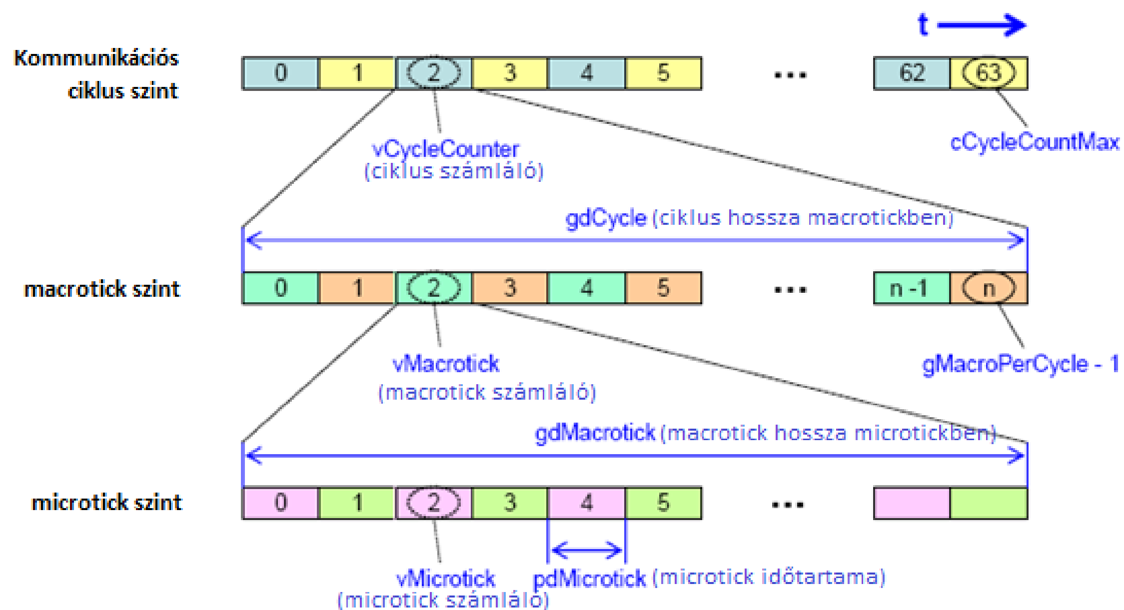


4.30. ábra: A FlexRay óra szinkronizáció mente

Az idő felépítése egy csomóponton belül ciklusokon, macrotick és microtick ütemeken alapszik. Egy makroticket egész számú mikrotick alkot és egy ciklus egész számú makroticket tartalmaz (4.31. ábra):

- A microtickek (mikró ütemek) olyan kontrollerspecifikus időegységek, melyeket a kontrollerhez tartozó oszcillátor állít elő. Időtartamuk kontrollertől függően változik. A csomópontok belső idejének finomságát adják.

- A macrotickek (makró ütemek) szinkronizálása klaszter alapú. Toleranciahatáron belül, egy makrotick időtartama azonos minden szinkronizált csomópontra a klaszteren belül. A macrotickek hossza a microtickek egész számú többszöröse, ahol a microtickek száma minden macrotickben más és más egy csomóponton belül. A microtickek száma egy macrotickben minden csomópontra más, és ezt a számot az oszcillátorok frekvenciája határozza meg. Annak ellenére, hogy bármely macrotick egész számú microticket tartalmaz, az átlagos hossza a makro ütemnek egy cikluson belül nem egész számú lesz.
- A ciklus egész számú macroticket tartalmaz, melyek száma azonos minden csomópontban egy cikluson belül, és ugyanaz marad ciklusról ciklusra. Bármely időpillanatban a csomópontoknak ugyanaz a ciklusszámuk.



4.31. ábra A macro- és microtickek felépítése

Az ismert eljárások, melyek az időszinkronizációt végzik, a csomópontok között fázis vagy frekvencia korrekciót használnak. A FlexRay ezen műveletek kombinációját használja. A következő feltételeknek kell teljesülnie:

- A frekvencia-, és fáziskorrekciónak hasonló módon kell zajlania minden csomópont esetén. A frekvenciakorrekciót az egész ciklus alatt kell végezni.
- A fáziskorrekciót a NIT alatt kell végrehajtani, mindig csak a páratlan ciklusokban, és be kell fejezni, mielőtt a következő ciklus kezdődne.
- A fázisváltozást a microtickek száma jelzi, melyeket hozzá kell adni a NIT fáziskorrekciós szegmenséhez, mely akár negatív is lehet. A kiszámítása minden ciklusban időt igényel, de a korrekciót csak minden páratlan ciklus végéhez kell hozzáadni. A fáziskorrekció kiszámítása egy ciklusban értékek mérésén alapul. Ezt a számítást nem lehet befejezni a NIT előtt, de elkezdeni már a dinamikus szegmensen vagy szimbólumablakon belül is lehet addig, ameddig a számítás visszacsatolása késleltetve van a NIT-ig. A számítást be kell fejezni, mielőtt a fáziskorrekciós fázis folytatódna.

- A frekvencia (rate) megváltozását microtickek száma jelzi, melyeket hozzá kell adni a kommunikációs ciklusban definiált microtickek számához, mely akár negatív is lehet. Értékét az óraszinkronizációs folyamat határozza meg, és csak egyszer kell kiszámolni dupla ciklusonként. Számítása időt igényel, páratlan ciklusokban a statikus szegmenset követően. A számítás páros és páratlan dupla ciklusokban történő értékek mérésén alapul. Ezt a számítást nem lehet befejezni a NIT előtt, de elkezdeni el lehet már a dinamikus szegmensen vagy szimbólumablakon belül addig, ameddig a számítás visszacsatolása késleltetve van a NIT-ig. A számítást be kell fejezni, mielőtt a következő páros ciklus elkezdődne.

5 Beágyazott rendszerek a járműiparban (Biztonságkritikus rendszerek)

5.1 Az autóipari beágyazott rendszerek szoftverfejlesztésének folyamata.

A mikrokontroller-, illetve processzoralapú beágyazott rendszerek valósidejű, vagy a hagyományos asztali környezetekhez közelálló operációs rendszereket futtathatnak. A mikrokontrollerek többnyire kisméretű (néhány kilobájt), speciális valós idejű operációs rendszert (RTOS, real-time operation system) futtathatnak, míg a nagyobb processzor alapú rendszerek többnyire összetettebb valós idejű operációs rendszereket (PharLap, VxWorks, stb.), vagy a hagyományos operációs rendszerek módosítatlan vagy módosított változatát (Windows, Linux, stb.) futtatják. Attól függően, hogy milyen eszközre, illetve milyen operációs rendszerre kell fejleszteni, számos programozási nyelvből lehet választani.

- A gépi kód a processzor számára önmagában értelmezhető bináris adat. Minden egyes utasítás egy bináris értékkel van leírva, melyet utasításkódnak (opcode) szoktak nevezni. Régebben volt igazán elterjedt a gépi kódban való programozás, manapság már csak szűk körben használatos.
- Az Assembly nyelven történő programozás közel áll a gépi kódban történő programozáshoz, ugyanakkor könnyebben átlátható a kód, mivel utasítás kódok helyett rövid szöveges utasítások vannak (mnemonics). A szöveges utasításokat az assembler fordítja le gépi kóddá, mely tulajdonképpen nem közvetlen átalakítás, de kevésbé összetett művelet, mint egy magasabb szintű programnyelv gépi kóddá alakítása.

  Az Assembly-t még mostanság is sok helyen alkalmazzák mikrokontrollerek, vagy akár processzorok programozására, ugyanakkor többnyire már csak ott használják, ahol egy utasítás nagy hatékonyságú elvégzése kulcs fontosságú, illetve esetlegesen az adott utasításnak nincs magasabb programnyelvű implementációja.

  További fontos szerep jut neki a hibakeresés során, mivel a legtöbb fejlesztő környezet képes a gépi kódot visszaalakítani Assembly nyelvre (Disassembly), így lehetővé téve a gépi kód könnyebb értelmezését, az esetleges program-, illetve fordítási hibák felismerését a magasabb szintű programnyelveknél.

  Komoly hátránya az Assembly nyelven írt programoknak, hogy a processzorhoz, pontosabban az adott architektúrához kötöttek.

- A C nyelv az egyik legelterjedtebb, a kisebb kontrollerekre talán a leginkább alkalmazott programozási nyelv. Már magas szintűnek minősülő, aránylag könnyen olvasható programozási nyelv, melyet többnyire összetett fordító alakít gépi kóddá.
- C++ és a JAVA objektumorientált programozási nyelvek. Elsősorban összetett programkódok esetén szokták alkalmazni, ahol fontos a modularitás. Kisméretű kontrollerek esetén többnyire nem kellően hatékonyak.
- Grafikus programozási nyelvek többnyire a fejlesztést hivatottak megkönnyíteni úgy, hogy nem programkódot kell írni, hanem grafikus módon, blokkok segítségével kerül összeállításra a program. Ilyen fejlesztői környezetek a Matlab/Simulink, NI

LabVIEW, stb. Hátrányuk, hogy ingyenes verzió ritkán érhető el ezen programokból, illetve hogy csak adott típusú kontrollereket támogatnak.

5.1.1 Valós idejű rendszerek követelményanalízise, modellezése és modellezési eszközei, HIL (Hardware in the Loop) és a SIL (Software in the loop) típusú szimulációk.

A különböző teszteket a fejlesztés különböző szakaszaiban szokták alkalmazni. Értelemszerűen a legalsó szint, melynél elkezdődik a tesztelés attól is függ, hogy egy adott fejlesztés milyen stádiumról indul például, hogy szükség van-e hardver kifejlesztésére, vagy csak hardverfejlesztésről van-e szó.

Mind szabályozó-, illetve vezérlőrendszerek (például blokkolás gátló rendszerek), mind pedig az intelligens szenzor rendszerek (például elektronikus akkumulátor szenzorok) esetében alkalmazni szokták ezen tesztelési módozatokat. A fejlesztés során az esetek többségében négy nagyobb tesztelési szintet szoktak megkülönböztetni, ezek a MIL, SIL, PIL és a HIL:

- A legalsó szint szokott lenni a „modell a hurokban”, azaz a MIL (model in the loop) tesztek. Ezek szoktak lenni a legelső fejlesztési fázis tesztlépései. Ekkor a szabályozó rendszerhez tartozó modell kerül letesztelésre, egy szintén modell alapú tesztkörnyezetben. Azaz ezen lépések még teljesen szoftver-, illetve modellalapúak. Ezen lépés esetén a legelterjedtebb fejlesztési környezet például a Matlab/Simulink illetve bizonyos esetekben az NI LabVIEW szokott lenni.
- A következő szint a „szoftver a hurokban” jellegű, azaz a SIL (software in the loop) tesztek. Ezeket akkor szokták végrehajtani, amikor a fejlesztendő rendszer elviekben már megfelelően működik, azaz a modell megfelelő, és a már a kész rendszerhez tervezett kódot kell vizsgálni. Lényegében ekkor már ismert a célhardver, amin a program futni fog, de még nem áll rendelkezésre, vagy nem lehet rajta tesztelni, éppen ezért a vizsgálatok még mindig a szoftveres környezetben zajlanak. A lényegi különbség a MIL tesztekhez képest, hogy míg ott csak voltaképpen az elvi működés (sokszor egy magasabb szintű és/vagy grafikus programozási nyelven implementált modell) kerül letesztelésre, addig a SIL tesztek esetében már az a programkód, ami már nagyon közel áll a végleges kódhoz, azaz ahhoz, ami a kiválasztott vagy megtervezett célhardverre fog kerülni. Sokszor a modellhez képest alacsonyabb programozási nyelven, például C vagy C++ nyelven kerülnek implementálásra ezen programok. Természetesen sokszor a magasabb nyelven elkészített modellből is lehetséges az alacsonyabb szintű programkód generálása, például NI LabVIEW vagy Matlab/Simulink esetén. Ezek a tesztek elsősorban már a kód helyességét, hibamentességét valamint megbízhatóságát vizsgálják.
- Az első tesztek, ahol már a hardveren is kipróbálásra kerül a kód, az a „processzor a hurokban” jellegű, azaz a PIL (Processor in the loop) tesztek. Ez már nem elsősorban a szoftvert, hanem az alkalmazni kívánt hardvert teszteli, pontosabban azt ellenőrzi, hogy az alkalmazni kívánt processzor, illetve központi egység megfelelő-e a megírt programkód helyes futtatására. Ugyanakkor azt is ellenőrzi, hogy a megírt programkód például kellően hatékony-e. Ennek akkor van elsődleges szerepe, ha a tervezett központi egységet nem lehet nagyobb teljesítményűre cserélni, illetve a

rendelkezésre álló memória mennyisége nem növelhető tovább. Ekkor a programkódon kell javítani, illetve optimalizálni kell azt. Itt elsősorban azt kell vizsgálni, hogy szükség van-e a hardver változtatására, és ekkor a tesztek többnyire még egy fejlesztői platformon futnak. Ezen tesztek során sokszor a korábbi tesztekhez használt, modell alapú tesztkörnyezet adja a jeleket. Ezen tesztekhez szintén gyakran alkalmazzák az NI LabVIEW/Veristand, valamint Matlab/Simulink szoftvereket.

- A legfelső szinten állnak a „hardver a hurokban”, azaz HIL (hardware in the loop) jellegű tesztek. Ekkor már a véglegeshez nagyon közel álló, vagy a véglegessel megegyező hardverkomponensen kell futtatni a fejlesztett programkódot. Nagyon fontos különbség az eddigi tesztekhez képest, hogy a HIL teszteket már valós időben szokás futtatni, míg az előzőek esetében ez többnyire nem lényeges. Ezen teszteknél lényegében kell egy pontos, valós időben futtatható modellje annak a környezetnek, ahol a tervezett rendszer működni fog. Ennek a modellnek kell szolgáltatnia a bemeneteket a tesztelendő eszköz felé, valamint reagálnia kell annak kimeneteire, azaz el kell hitetnie az eszközzel, hogy az éppen a végleges környezetében működik.

6 Időkezelés és adatátvitel

Az elosztott rendszerek jellemző problémája az időkezelés. A kommunikáció véges volta miatt előfordulhat, hogy ha két csomópont egymást követő eseményéről (az esemény a rendszer-állapot detektálható, pillanatszerű változása) tájékoztatást ad két másik csomópontnak, akkor az üzenetek megérkezési sorrendje el fog térni az események időbeni sorrendjétől. Ebben az esetben a csomópont a korábban bekövetkezett eseményről később szerez tudomást. A probléma feloldásához fontos a csomópontok közti megegyezés.

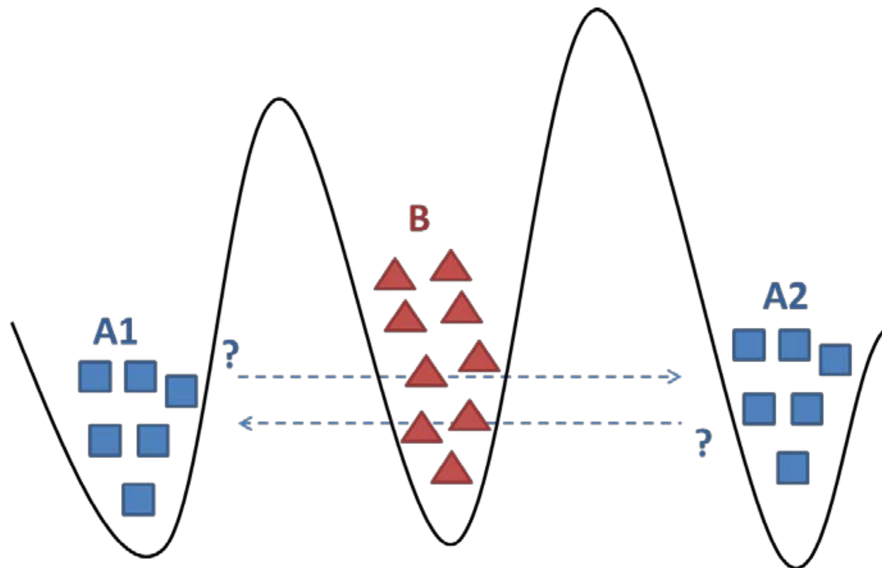
Az időkezelés rendkívül fontos a valós idejű rendszerek esetében, mivel egy meghatározott időn belül el kell kezdeni az esemény feldolgozását. A követelmények szigorúsága alapján két féle valós idejű rendszert különböztethetünk meg. A „Hard real-time” rendszerek esetében szigorú követelmények vannak előírva, és a kritikus folyamatok meghatározott időn belül feldolgozásra kell, hogy kerüljenek. „Soft real-time” rendszer esetében a követelmények kevésbé szigorúak és a kritikus folyamatokat a rendszer mindössze nagyobb prioritással dolgozza fel.

Lehetetlenségi tétel (Two Generals' Problem)

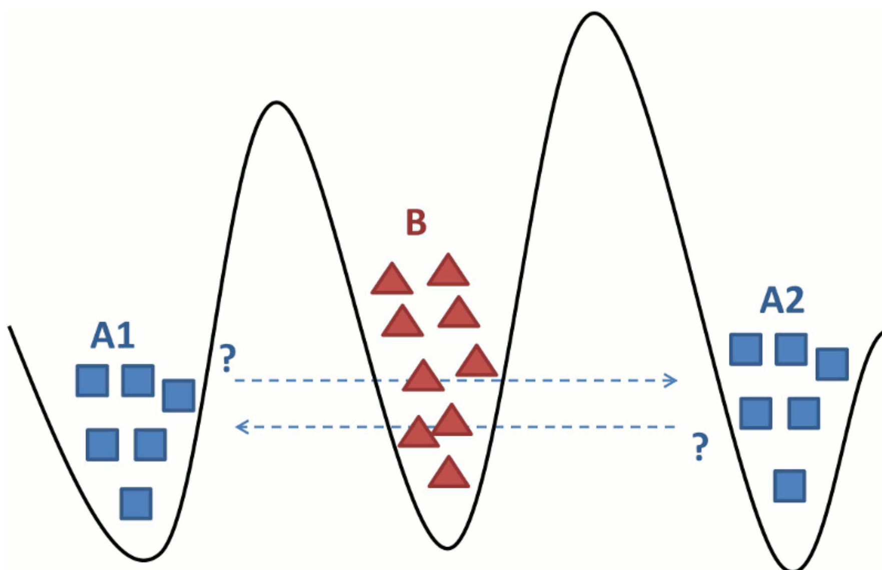
A biztonságos üzenetküldés és kommunikáció alapvető követelménye, hogy a küldő meggyőződjön arról, hogy az elküldött üzenetet a vevő fél megkapta, vagyis mindkét fél biztos legyen abban, hogy egy adott művelet végrehajtódott. Itt a veszélyt az jelenti, ha manipuláció vagy hibás átvitel eredményeként az egyik fél úgy gondolja, hogy a művelet sikeresen végbement. A probléma az, hogy hogyan tud a művelet sikeréről meggyőződni a küldő fél. Látszólagos egyszerűsége ellenére ezt a követelményt nehéz biztosítani.

„Szövetség a völgy felett” (Two Generals' Problem) probléma a megegyezés nehézségét szemléleti. A gondolatkísérlet célja, hogy bemutassa a buktatókat és a tervezési nehézségeket egy koordinált akció megszervezésénél, ha a kommunikációt megbízhatatlan csatornán kell bonyolítani. Egy völgyben táborozott 5000 frankus katona, míg a völgy két oldalán a 3-3 ezer burkus és bergengóc katona. Burkus király (A1) és Bergengóc király (A2) szövetségesek, egyszerre akarják megtámadni a völgyben táborozó frankusokat, ezért burkus király hírnököt küld az ellenség táborán át, hogy „holnap négykor támadunk”.

A két szövetséges hadvezér csak futárok segítésével az ellenséges vonalakon keresztül tud kommunikálni egymással. A futárokat azonban elfoghatja az ellenség, így nem biztos, hogy az üzenet megérkezik. A kérdés az, hogy megoldható-e valamilyen kommunikációs módszerrel az, hogy a két hadvezér meg tud-e egyezni a támadás időpontjában. Ha A1 csomópont üzenetet küld A2-nek, az üzenet megérkezik, de A2 nem lehet biztos benne, hogy A1 tud az üzenet sikeres megérkezéséről, ezért küld egy nyugtázó üzenetet. Ezt A1 megkapja, de ő sem lehet benne biztos, hogy A2 tud a nyugtázó üzenet sikeres megérkezéséről. Így a végtelenségig lehetne hírnököket küldözgetni a két tábor között, de úgysem lenne 100%, hogy mind a két oldal biztos arról, hogy megegyeztek... (6.1. ábra)



6.1. ábra Két tábornok problémája



6.2. ábra Két tábornok problémája animáción bemutatva

Indirekt bizonyítással és teljes indukcióval egyszerűen belátható, hogy ilyen megoldás nem létezik. Tételezzük fel, hogy véges „n" lépésben („n" darab futár küldése után) meg tudnak egyezni a hadvezérek. Ekkor viszont az „n"-edik lépésben küldött futár elfogásának esetén arra a következtetésre kellene jutnunk, hogy már az (n-1)-ik lépésben is tudniuk kellett volna a hadvezéreknek a támadás időpontjáról. Véges lépésben ellentmondásra jutunk, ha az (n-1)-ik lépésre ugyanezt a gondolatmenetet alkalmazzuk. Ez azt jelenti, hogy az első futár küldésekor tudni kellett volna a támadás időpontját. A kiindulási helyzet viszont az volt, hogy nem tudják a hadvezérek a támadás időpontját.

A történelmi példa alapján láthatjuk, hogy a legrosszabb esetet feltételező üzenetvesztés esetén nem létezik olyan biztonságos nyugtázott üzenetküldés, amely során mindkét fél meggyőződhet arról, hogy egy egyeztetés sikeres volt. Ha valamilyen természetes, nem rosszindulatú üzenetvesztést tételezünk fel (például az átviteli csatornán lévő zaj miatt), akkor az üzenet továbbításának sikerességét valamekkora valószínűséggel jellemezhetjük.

Abban az esetben, ha egy üzenetküldés biztonságos nyugtázását valamilyen valószínűséggel meg tudjuk oldani, akkor az a gyakorlatban biztonságosan megoldható probléma.

Ha rosszindulatú, intelligens támadást feltételezünk, akkor a támadó az üzenet egyeztetésének módszerét is ismeri. Ebben az esetben a Bizánci problémánál látott bizonyítás alapján egy támadó bármely kifinomult protokoll esetén elnyelhet bizonyos üzeneteket. Így valamelyik felet kétségek között tudja tartani. Ebben az esetben nem létezik elméleti és nem létezik gyakorlati megoldás sem. Ez a probléma csak mindkettő fél által hitelesített harmadik fél bevonásával oldható meg. Ha a kommunikáció nem 100%-ig biztos, akkor lehetetlen a 100%-os megegyezés.

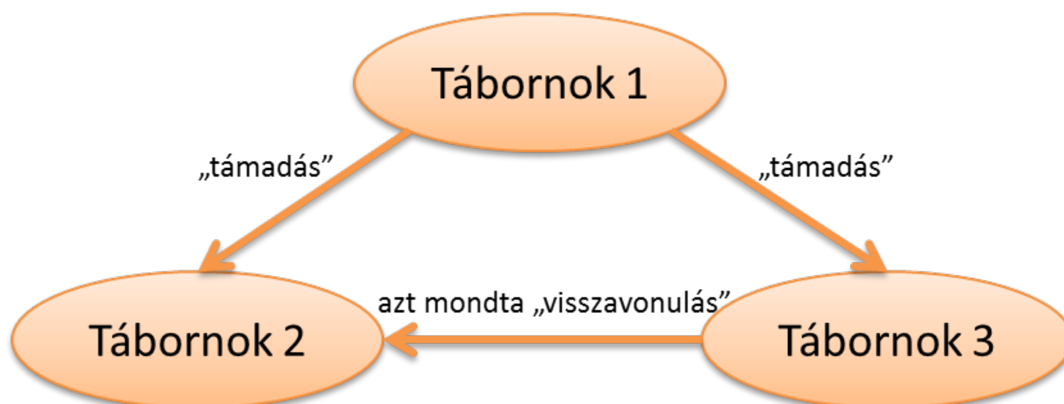
Lehetetlenségi tétel: Véges meghibásodású csatornáról nem lehet hibátlan kommunikációt feltételezni.

Bizánci tábornokok problémája (Byzantine generals problem)

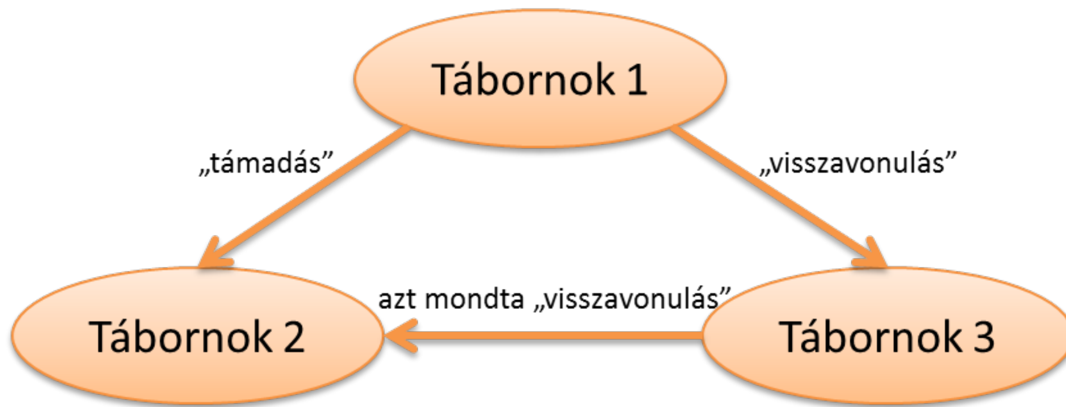
Bizonyos biztonságkritikus rendszereknél több érzékelőt, vezérlőt, esetleg több számítógépet használnak ugyanannak a jelnek a mérésére, a folyamatok vezérlésére. Erre a redundanciára azért van szükség, hogy egy részegység meghibásodása esetén is működőképes maradjon a rendszer. A probléma akkor keletkezik, ha egy rendszernél egy érzékelő teljesen más értéket mér, vagy egy vezérlő máshogyan dönt, mint a többi.

A probléma bemutatására a bizánci tábornokok problémáját használják, mely így hangzik: N tábornok táborozik a seregeivel (1000, 2000, 3000 és 4000 fős seregek) egy város körül, amit meg akarnak ostromolni. (A várost 5000 fős sereg védi.) Tudják, hogy elég sokan vannak ahhoz, hogy ha az összesített haderejüknek legalább a felével sikerül egyszerre támadniuk, akkor győzni fognak. Ha azonban nem sikerül pontosan koordinálniuk a támadás időpontját, akkor szétszóródnak a seregeik, és csatát veszítenek. Sejtik továbbá azt is, hogy vannak köztük árulók is, akik hamis üzeneteket küldenek majd. Mivel csak futárok útján tudnak kommunikálni egymással, így nem tudják ellenőrizni az egyes üzenetek hitelességét. Hogyan egyezhet meg a közös támadás időpontjáról egy ilyen nagy csoport egy bizalmas központi hatóság nélkül – főleg akkor, ha még zavarkeltő árulókkal is meg kell küzdeniük?

Az adatok alapján látszik, hogy a támadó seregek csak összefogással győzhetik le a várost védőket. Jelen esetben tehát az adat érvényességével van a baj, nem a kommunikációval.



6.3. ábra Tábornok 3 az áruló



6.4. ábra Tábornok 1 az áruló

Mindegyik csomópont (tábornok) elküldi a többieknek, hogy mennyi katonája van. Feltételezzük, hogy a 3-as hazudik, minden üzenetben mást mond (6.3. ábra),(6.4. ábra), jelöljük ezeket x, y, z-vel. Az egyes csomópontok a következő üzeneteket kapták:

1. (1, 2, x, 4), 2. (1, 2, y, 4), 3. (1, 2, 3, 4), 4. (1, 2, z, 4)

Ezután elküldik egymásnak a kapott üzeneteket, 3-as ismét mindenkinek mást hazudik:

1. a következő üzeneteket kapja: (1, 2, y, 4), (a, b, c, d), (1, 2, z, 4)

2. a következő üzeneteket kapja: (1, 2, x, 4), (e, f, g, h), (1, 2, z, 4)

4. a következő üzeneteket kapja: (1, 2, x, 4), (1, 2, y, 4), (i, j, k, l)

Így már kiszűrhető, hogy 1000 + 2000 + 4000 = 7000 katona biztosan van, tehát megkezdhetik a támadást. A helyes döntés meghozatalához „m" megbízhatatlan egység esetén 3m+1 iterációra van szükség. Ez a szabály a redundáns rendszerek esetében is használható.

A Bitcoin megoldást ad erre a problémára, a közmegegyezés kialakításának egyik legnagyobb kihívására. A Bitcoin megoldása a következő: minden tábornok elkezd dolgozni egy olyan matematikai probléma megoldásán, ami statisztikailag 10 percet vesz igénybe akkor, ha mindannyian munkálkodnak rajta. Amint az egyikük megtalálja a megoldást, nyomban elküldi azt az összes többinek is. Ezt követően mindenki ezzel a megoldással dolgozik tovább, ebből kiindulva keresi a következő megoldást, ami megint csak tíz percet vesz igénybe. Minden tábornok mindig az általa ismert leghosszabb megoldássorozattal dolgozik, azt bővítve tovább. Ha pedig már van egy 12-szeresen kibővített megoldásuk, akkor mindannyian teljesen biztosak lehetnek benne, hogy egyetlen áruló sem hozhatott létre sehogyan sem egy ilyen hosszú megoldássorozatot anélkül, hogy ne rendelkezne mindannyiuk összesített számítókapacitásának legalább a felével. A 12 blokkos lánc léte minden résztvevő számára egyértelműen bizonyítja, hogy a többségük tisztességesen kivette a részét annak a létrehozásából. Ezt nevezzük munkabizonyíték-rendszernek.

Mindez azt jelenti, hogy a vitathatatlanul hiteles közmegegyezés kialakítását a rendelkezésre álló számítási erőforrások korlátozottsága teszi lehetővé. Ahhoz, hogy sikeresen támadható legyen a rendszer, egy támadónak nagyobb számítási kapacitással kellene rendelkeznie annál, mint amennyit a tisztességes csomópontok birtokolnak együttesen.

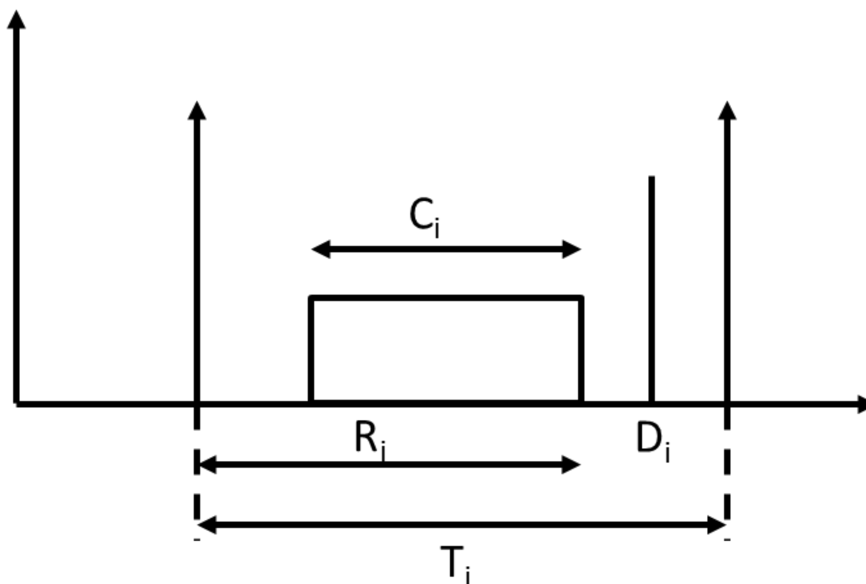
Ütemezés

A feladatok (task) közül a valós idejű operációs rendszerek számára kritikus az ütemezés és az erőforrásokkal való helyes gazdálkodás megvalósítása. Mivel minden rendszer valamilyen periféria segítségével kommunikál a környezetével, ezért fontos e perifériáknak a valós idejű rendszer követelményeinek megfelelő módon történő kezelése. Az ütemezés és az erőforrásokkal való gazdálkodás azért kiemelt fontosságú, mert egy-egy esemény kezelésekor a válaszidő betartásához az eseményt lekezelő utasítássorozatot végre kell hajtani. Az utasítássorozat lefutása erőforrásokat igényel, melyeket az operációs rendszernek biztosítani kell. Ez úgy valósítható meg a leggyorsabban, ha az operációs rendszer folyamatosan rendelkezik szabad erőforrásokkal, melyeket oda tud adni az időkritikus folyamatoknak.

6.1.1 Task tulajdonságai és az ütemezés kapcsolata

Az ütemezés során a folyamatok két legfontosabb tulajdonsága a kritikusság és az időzítési viszonyok. Nem elég csupán az időzítési viszonyokat figyelembe venni. A kommunikációs, szinkronizációs tényezőkre, a precedenciára és a kölcsönös kizárásra is figyelni kell. Az időzítési viszonyokat az alábbi attribútumokkal írhatjuk le (6.5. ábra):

- C_i az i-edik task végrehajtási ideje (computation time, kiszolgálási idő)
- D_i az i-edik task végrehajtásának határideje (dead time)
- R_i az i-edik task válaszideje (response time),
- T_i az i-edik task periódusideje, ennek letelte után futtatható a következő task



6.5. ábra Task futásának fontosabb időpontjai

6.1.2 Ütemezők típusai

A CPU ütemezésnek különböző szintjeit tudjuk megkülönböztetni:

- Hosszú távú (long term) ütemezés, vagy munka ütemezés
- Középtávú (medium term) ütemezés
- Rövidtávú (short term) ütemezés

Nem minden általános célú operációs rendszerben van mindegyik ütemezés megvalósítva. A nagy rendszerekben mind három típus jelen lehet. A kis vagy közepes rendszerekben egy, legfeljebb két ütemező van. Amikor több mint egy ütemező van, nagyon fontos az együttműködésük.

A hosszú távú ütemező a nagy erőforrás igényű alacsony prioritású folyamatokat választja ki, amit arra használhatunk, hogy az alacsony aktivitású időszakokban dolgoztassuk az erőforrásokat. A hosszú távú ütemező elsődleges célja biztosítani az alacsony prioritású folyamatok egyenletes terhelését. A hosszú távú ütemező használata rendszer-, és terhelésfüggő, de sokkal ritkább, mint a többi ütemezőé. Feladata, hogy a háttértáron várakozó, még el nem kezdett munkák közül meghatározza, hogy melyek kezdjenek futni, a munka befejezésekor ki kell választania egy új elindítandó munkát. A hosszú távú ütemezést végző algoritmusnak ezért ritkán kell futnia.

A középtávú ütemezés az időszakos terhelésingadozásokat hivatott megszüntetni, hogy a nagyobb terhelések esetében ne legyenek időtúllépések. A középtávú ütemező algoritmus ezt úgy oldja meg, hogy bizonyos (nem időkritikus) folyamatokat felfüggeszt, illetve újraaktivál a rendszer terhelésének a függvényében. Folyamat felfüggesztése esetén a folyamat a háttértáron tárolódik, az operációs rendszer elveszi az erőforrásokat, melyeket csak az újraaktiválásakor ad vissza a felfüggesztet folyamatnak.

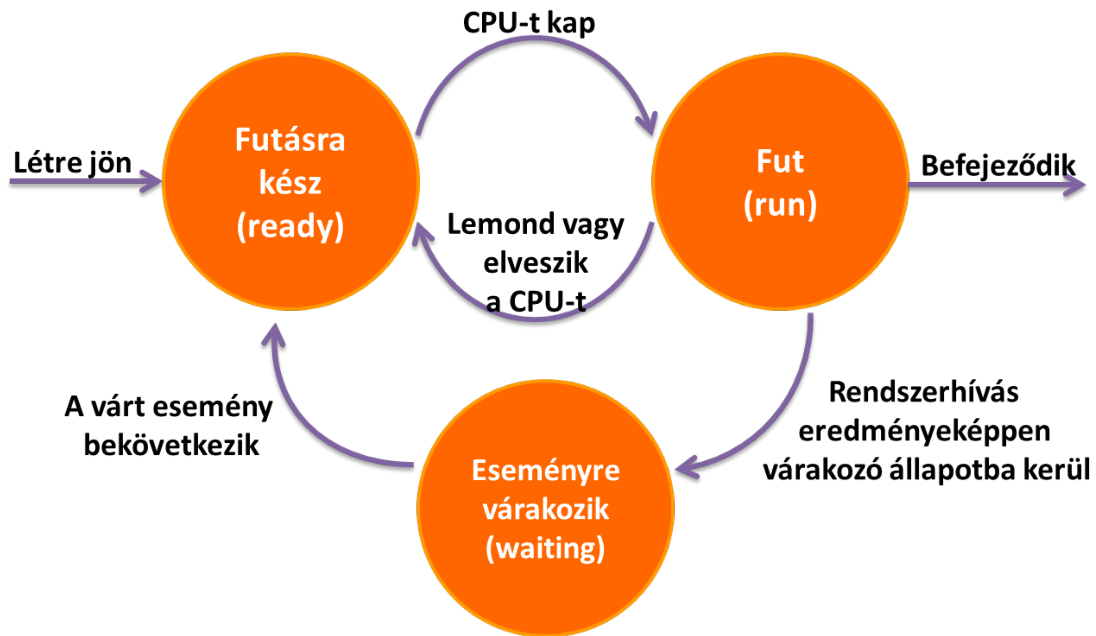
A rövidtávú ütemezés feladata, hogy kiválassza, hogy melyik futásra kész folyamat kapja meg a CPU-t. Fő feladata maximalizálni a rendszer teljesítményét bizonyos feltételek kielégítése mellett (határidők). A rövidtávú ütemezést végző algoritmus gyakran fut le, ezért gyorsan kell lefutnia. Mivel gyakran lefut az algoritmus, ezért az operációs rendszer mindig a memóriában tartja az ütemező kódját. Az operációs rendszerek magja tartalmazza az ütemezőt.

Az általános célú és a valósidejű operációs rendszerek a CPU ütemezésben különböznek leginkább egymástól. Ennek az oka az, hogy a valósidejű operációs rendszereknek az eseményeket meghatározott időn belül le kell reagálnia, egy általános célú operációs rendszer esetében nincsenek ilyen jellegű követelmények.

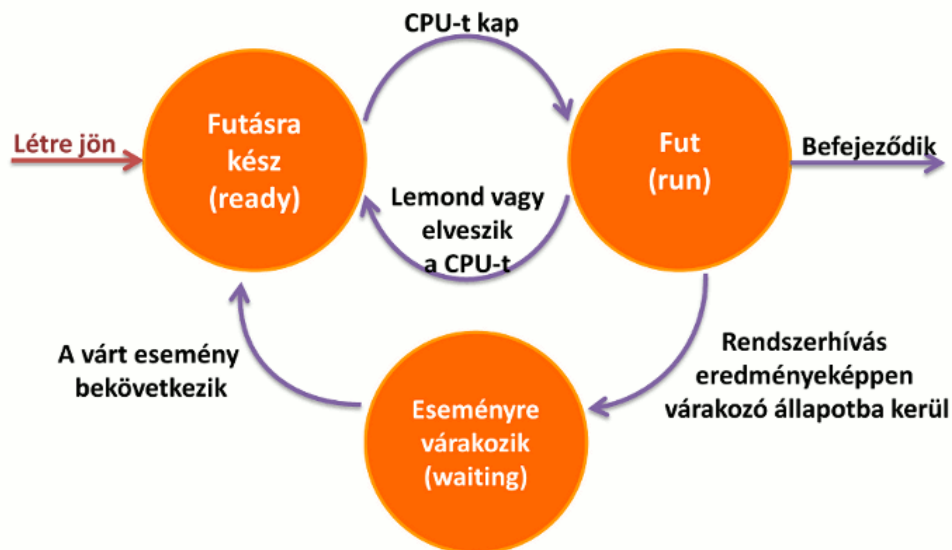
Az ütemezéssel kapcsolatban a következő alapfogalmakat értelmezhetjük:

- Task: Önálló részfeladat.
- Job: A task-ok kisebb, rendszeresen végzett feladatai.
- Process: A legkisebb futtatható programegység, egy önálló ütemezési entitás, amelyet az operációs rendszer önálló programként kezel. Van saját (védett) memóriaterülete, mely más folyamatok számára elérhetetlen. A task-okat folyamatokkal implementálhatjuk.
- Thread: Saját memóriaterület nélküli ütemezési entitás, az azonos szülőfolyamathoz tartozó szálak azonos memóriaterületen dolgoznak.
- Kernel: Az operációs rendszer alapvető eleme, amely a task-ok kezelését, az ütemezést, és a task-ok közti kommunikációt biztosítja. A kernel kódja hardver függő (device driver) és hardware független rétegekből épül fel. A hardware függő réteg új processzorra és eszközökre történő adaptálását az operációs rendszer portolásának nevezzük.

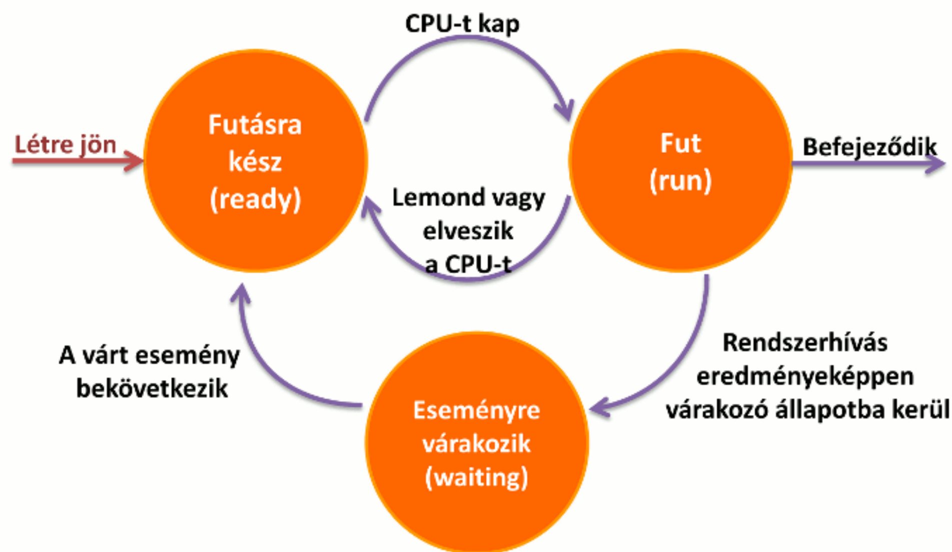
- CPU löket (CPU burst): A folyamatnak csak CPU és az operatív tár kell
- Periféria löket (I/O burst): Perifériás átvitelt hajt végre a folyamat, nincsen szükség CPU-ra



6.6. ábra Folyamat állapotai ütemezés szempontjából



6.7. ábra Megszakításos ütemező



6.8. ábra Nem megszakításos ütemező

Ütemezés során a folyamatokkal a következő esemény következhet be:

- A futó folyamat várakozni kényszerül (Például: I/O-ra, erőforrásra). A task egy meghatározott eseményre várakozik. (Ez rendszerint valamilyen I/O művelet szokott lenni.)
- A futó folyamat befejeződik.
- A futó folyamat lemond a CPU-ról.
- A futó folyamattól az operációs rendszer elveszi a CPU-t. A task-ot megszakították, vagy a megszakítás kezelő rutin éppen megszakítja a folyamatot.
- A folyamat aktiválódik, futásra késszé válik. A futásra kész állapotot jelöli. Fontos a task prioritási szintje és az is, hogy az éppen aktuálisan futó task milyen prioritási szinttel rendelkezik, ezek alapján dönti el az ütemező, hogy elindítja-e a taskot.

A task-ok állapotát és tulajdonságait a Task Vezérlő Blokk (Task Controll Block – TCB) írja le, amely a memóriában lévő adatszerkezet. Fontosabb tagjai a következők:

- Task ID: Egy egész szám, amely a task-ot azonosítja.
- Context: Program Counter: a regiszterek és flag-ek elmentett értékei. (A task futásának helyreállításához szükségesek ezek az információk.)
- Top of Stack: Egy mutató, amely megadja a task-hoz tartozó verem tetejét
- Status: Egy egész szám, amely utal a task aktuális státuszára.
- Priority: A prioritás aktuális értéke, amely a futás közben megváltoztatható.
- I/O Information: Leírja, milyen perifériákat és I/O-kat foglalt le és használ a task. A nem használt perifériákat minden esetben fel kell szabadítani.

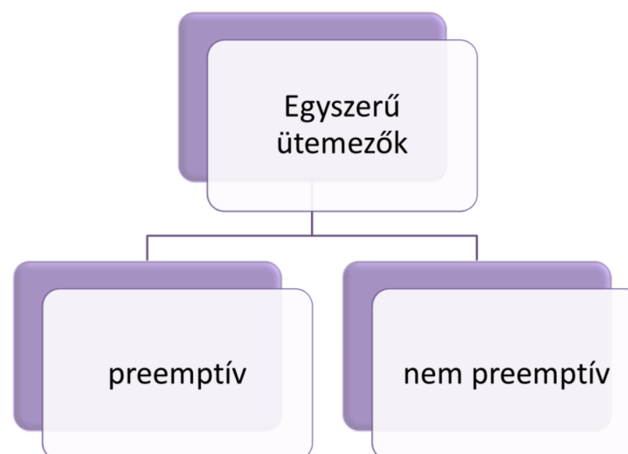
Bármilyen ütemezési stratégiát is alkalmazzunk, a folyamatmenedzselő rendszernek a megszakítások kezelésével is kell foglalkoznia. Ezek a megszakítások lehetnek hardveresek és szoftveresek egyaránt. A megszakítás környezetváltást (context switch) követel. A futó folyamat megszakad, és a megszakítás kezelő egy gyorsan végrehajtható kódot futtat, amely a következő folyamat futásához szükséges értékeket tölti be a regiszterekbe. Ha a megszakítás kiszolgálása befejeződött, akkor vagy a megszakított folyamat futása folytatódik, vagy az ütemező indul el, és megállapítja a következő végrehajtandó folyamatot.

Az ütemezési algoritmusokat csoportosíthatjuk felépítésük és működésük alapján. A különböző operációs rendszerek használhatóságát nagyban befolyásolja az ütemező algoritmus működése. A klasszikus ütemezési algoritmusok közül a következőket tárgyaljuk:

- Egyszerű algoritmusok
    - Legrégebben várakozó (First Come First Served, FCFS):
    - Körforgó (Round-Robin, RR)
- Prioritásos algoritmusok
    - Statikus prioritás
    - Legrövidebb (löket)idejű (Shortest Job First, SJF)
    - Legrövidebb hátralévő idejű (Shortest Remaining Time First, SRTF)
    - Legjobb válaszarányú
- Többszintű algoritmusok
    - Statikus többszintű sorok (Static Multilevel Queue, SMQ)
    - Visszacsatolt többszintű sorok (Multilevel Feedback Queues, MFQ)
- Többprocesszoros ütemezés

6.1.2.1 *Egyszerű ütemezők*

Az egyszerű ütemezési algoritmusok az ütemezésre kerülő folyamatokat minden esetben egyenlőnek, azonos prioritásúnak tekintik.



6.9. ábra Egyszerű ütemezők típusai

6.1.2.1.1 Legrégebben várakozó (First Come First Served)

A legegyszerűbb algoritmus, a feladat leíróra mutató referenciákat tároló FIFO (First Input First Output) sor az implementáció. Definíció szerint nem preemptív (nemmegszakításos), de I/O-ra várhat. Az átlagos várakozási idő nagy lehet, és erősen függ a feladatok hosszától, és a

CPU és I/O löket (burst) nagyságoktól. Ez egyben átlagosan nagy válaszidőt is jelent, on-line felhasználókat is kiszolgáló rendszerek számára nem alkalmas. De kis adminisztrációs overhead jellemzi, a minimális számú kontextusváltás miatt. Az új folyamatok a várakozási sor végére kerülnek, mindig a sor elején álló folyamat kezd futni, a folyamatok nem szakíthatóak meg. (Nem preemptív az ütemező, így valós idejű rendszerhez nem használható.) Az algoritmus előnye az, hogy egyszerűen megvalósítható. Az algoritmus hátránya, hogy egy hosszú ideig futó folyamat feltartja az egész rendszert (Konvojhatás)

6.1.2.1.2 Körforgó (Round-Robin – RR)

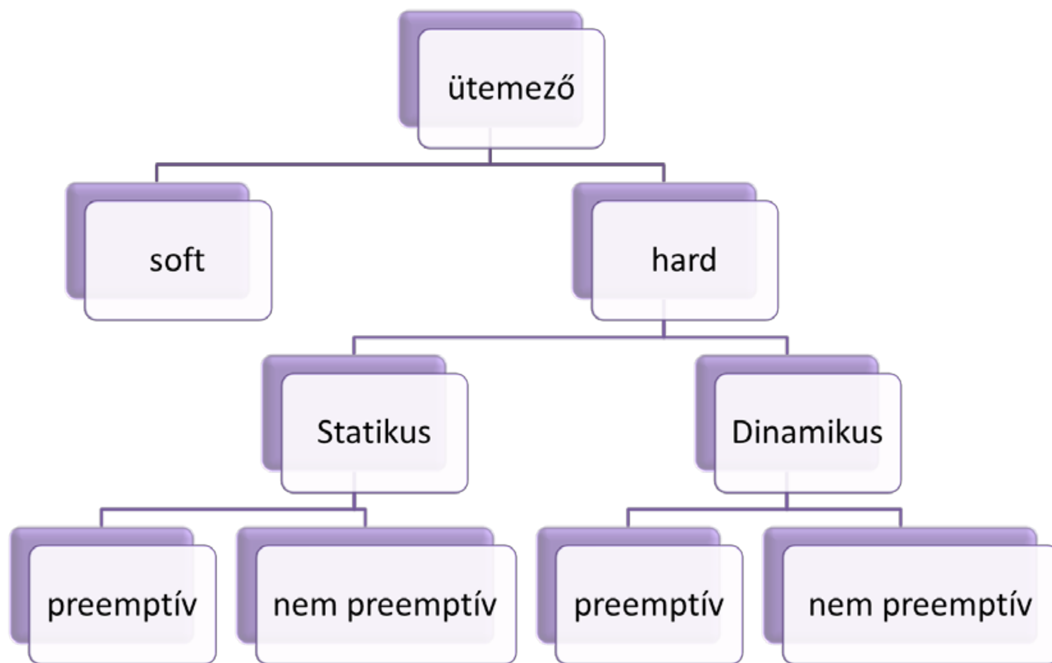
Időosztásos rendszerek számára találták ki az egyszerű FCFS ütemezés problémáinak kijavítására. Kedvezőbb az on-line felhasználók számára, mivel jobb az átlagos válaszideje a FCFS-nél. (Adott időnként garantáltan vált, függetlenül a feladattól.) Az algoritmus tehát az időosztásos operációs rendszerek algoritmusainak alapja. Csak időszeleteket kapnak a folyamatok (time slice), amelyek után az ütemező átadja a vezérlést egy másik folyamatnak, így az algoritmus preemptív módon üzemel. Abban az esetben, ha a CPU löket kisebb, mint az időszelet, akkor a folyamat lefut és átadja a vezérlést a soron következő folyamatnak. Ha a CPU löket nagyobb, mint az időszelet, akkor az időszelet lejárta után felfüggesztésre kerül a futó folyamat, és az ütemező átadja a vezérlést a soron következő folyamatnak.

Túl hosszú időszelet esetén az algoritmus ugyanúgy viselkedik, mint az FCFS algoritmus, míg túl kicsire választás esetén megnő a környezetváltások száma, ami a rendszer hasznos teljesítményét jelentősen lerontja. A statisztikai vizsgálatok alapján az időszelet hosszát úgy kell megválasztani, hogy a CPU-löketek kb. 80%-a legyen rövidebb az időszeletnél.

6.1.2.2 Prioritásos ütemezők

A folyamatok futási sorrendjét a prioritásuk határozza meg. A folyamatokhoz prioritást rendelünk, a prioritását különböző attribútumok alapján határozzák meg. Az ütemező a legnagyobb prioritású folyamatnak osztja ki a CPU-t. Ha egyenlő a folyamatok prioritása, akkor az egyenlő prioritású folyamatok között valamelyik egyszerű ütemezési stratégiát használjuk a sorrend eldöntésére. A prioritás meghatározása szempontjából lehet:

- statikus prioritású rendszer (előre meghatározott prioritás)
    - Tervezési időben teljesen meghatározott, hogy milyen feladatok és mikor futnak.
    - Legrosszabb esetre tervezés.
    - Speciális, többnyire biztonságkritikus beágyazott rendszerekben alkalmazzák.
- dinamikus prioritású rendszer (futási időben meghatározott prioritás)
    - A gyakorlatban használt algoritmusok ilyenek.
    - Dinamikus erőforrás kihasználás.
    - Tervezési időben nehezen vizsgálhatók.



6.10. ábra Prioritásos ütemezők típusai

A statikus ütemező az ütemezési döntéseket fordítási időben hozza meg. Egy diszpécser táblázatot generál, amely alapján a program végrehajtásra kerül. A dinamikus (vagy on-line) ütemező az ütemezésre vonatkozó döntéseket futás közben hozza meg. Ezeken belül megkülönböztetünk még preemptív, és nem-preemptív eseteket, azaz amikor az ütemezett folyamat megszakítható, ill. amikor nem szakítható meg. Nem preemptív algoritmus, amely a legrégebben várakozó folyamatot választja ki futásra. Megvalósítása igen egyszerű, a futásra kész folyamatok egy várakozási sor végére fűződnek fel, az ütemező pedig mindig a sor legelején álló folyamatot kezdi futtatni. Ennél az algoritmusnál igen nagy lehet az átlagos várakozási idő, mivel egy-egy hosszú CPU-löketű folyamat feltartja a mögötte várakozókat (ezt hívjuk konvoj hatásnak).

Először a kooperatív multitask-ot valósították meg nagy gépes környezetben. A működési elve és alapötlete a kooperatív algoritmusoknak az, hogy egy adott program vagy folyamat lemond a processzorról, ha már befejezte a futását vagy valamilyen I/O műveletre vár. Ez az algoritmus addig működik hatékonyan, amíg a szoftverek megfelelően működnek (nem kerülnek végtelen ciklusba), és lemondanak a CPU-ról. Ellenkező esetben az egész rendszer stabilitását képes lecsökkenteni vagy akár képes az egész rendszert lebénítani. A kooperatív algoritmus ezért soha nem fordul elő valós idejű operációs rendszerek esetében.

A preemptív algoritmusok esetében az operációs rendszer részét képező ütemező algoritmus vezérli a programok/folyamatok futását. A preemptív multitask esetén az operációs rendszer elveheti a folyamatoktól a futás jogát és átadhatja más folyamatoknak. A valós idejű operációs rendszerek ütemezői minden esetben preemptív algoritmusok, így bármely program vagy folyamat leállása nem befolyásolja számottevően a rendszer stabilitását. Az ütemező algoritmusok az operációs rendszerek rendeltetése alapján más-más rendszerjellemzőkre vannak optimalizálva. Az ütemezési algoritmusok teljesítményét a következő szempontok alapján tudjuk osztályozni:

- CPU kihasználtság (CPU utilization): Azt mondja meg, hogy a CPU az idejének hány százalékát használja a folyamatok utasításainak végrehajtására.
- CPU üres járása (Idle): A CPU idejének hány százalékában nem hajt végre folyamatot. (Ilyenkor indíthatóak hosszú távú (long term) ütemezésben szereplő folyamatok.)
- Átbocsátó képesség (Throughput): Az operációs rendszer időegységenként hány folyamatot futtat le.
- Körülfordulási idő (Turnaround time): A rendszerbe helyezéstől számítva mennyi idő alatt fejeződik be egy process
- Várakozási idő (Waiting time): Egy munka (vagy folyamat) mennyi időt tölt várakozással
- Válaszidő (Response time): Időosztásos (interaktív) rendszereknél fontos, azt mondja meg, hogy mennyi a kezelői parancs/beavatkozás után a rendszer első válaszáig eltelt idő.

Az ütemezési algoritmusokkal szembeni követelményeket különbözőképpen tudjuk csoportosítani. Rendszerenként változhat az, hogy a megvalósításkor melyik követelményt választják fontosnak és melyiket kevésbé. Az algoritmusokkal szemben támasztott fontosabb követelmények a következők:

- Optimális: Legyen optimális a rendszer viselkedése, azaz valamilyen előre meghatározott szempontok figyelembe vételével működjön a rendszer.
- „Igazságos”: Ne részesítse előnyben azonos paraméterekkel rendelkező process-ek közül semelyiket sem.
- Prioritások kezelése: Legyen képes az algoritmus arra, hogy a process-eket, folyamatokat a prioritásuk alapján egymástól megkülönböztessen.
- Ne „éheztesse ki” a folyamatokat: Minden process kapjon valamennyi processzoridőt, ezáltal biztosítva azt, hogy folyamatos legyen a futás
- Viselkedése legyen megjósolható: Minden esetben legyen a rendszer viselkedése előre kiszámítható, hogy a mérnökök előre modellezni tudják a rendszer viselkedését.
- Minimális rendszeradminisztrációs idő.
- Graceful degradation: Ez a rendszer túlterhelése esetén fontos szempont, mert a rendszer viselkedésének szempontjából az a fontos, hogy „fokozatosan romoljon le” a rendszer teljesítménye. A valós idejű operációs rendszerek esetében kritikus milyen csökkenés engedhető meg, mert a rendszernek tartani kell a valós idejű rendszer specifikációjában meghatározott időket. Ha a rendszer terhelése eléri az könyökkapacitást, akkor utána viselkedése megváltozik, a tovább növekvő terhelésre már egyre rosszabb működéssel reagál (overhead).

Prioritásos ütemező algoritmusoknál a folyamatokhoz az ütemező hozzárendel egy prioritásértéket, és a legnagyobb prioritású folyamat kapja meg a futás jogát. Ezeknél az algoritmusoknál megkülönböztethetünk statikus, és dinamikus prioritásos algoritmusokat. A statikus prioritásos algoritmusoknál a folyamatok kiéheztetése léphet fel, ezért a folyamatokat öregíteni (aging) kell.

- Legrövidebb (löket)idejű (Shortest Job First – SJF): Az algoritmus a dinamikus prioritásos algoritmusok közé tartozik, a várakozó folyamatok közül a legrövidebb löketidejűt indítja el.
- Legrövidebb hátralévő idejű (Shortest Remaining Time First – SRTF): Az algoritmus szintén dinamikus prioritásos algoritmus. Ha egy új folyamat érkezik, akkor az ütemező megvizsgálja a futásra kész folyamatok hátralévő löketidejét, és a legrövidebb hátralévő idejű folyamatot indítja el.
- A legjobb válaszarányú algoritmus (Highest Response Ratio - HRR): Dinamikus prioritásos algoritmus. A prioritásos algoritmusok nagy hátránya a kiéheztetés veszélye. (Vagyis, hogy a kis prioritású folyamatok elől a nagyobb prioritásúak „ellopják" a CPU-t.) Ennek kivédése az öregítés (aging) segítségével történhet, ezáltal a rendszer a régóta várakozó folyamatok prioritását fokozatosan növeli. Ezt az elvet alkalmazza az SJF-ből kiindulva a HRR-algoritmus. A folyamatok kiválasztásánál a löketidő mellett a várakozási időt is figyelembe veszi. A prioritás alapjául a *(Löketidő + k*Várakozási idő)/Löketidő* összefüggés szolgál (*k* egy alkalmasan megválasztott konstans).

6.1.2.3 Többszintű algoritmusok

A többszintű algoritmusok esetében a folyamatok több sorban várakoznak (például: rendszer, megjelenítés, batch folyamatok, stb.). Minden sorhoz prioritást rendel az ütemező algoritmus. A sorokon belül különböző kiválasztási algoritmusok is használhatóak. A többszintű algoritmusoknak kettő fő típusa van: statikus többszintű sorok (folyamat nem kerülhet át másik ütemezési sorba) és a visszacsatolt többszintű sorok (folyamat átkerülhet másik ütemezési sorba). A hatékony többprocesszoros ütemezés a mai processzorok esetében elengedhetetlen, mivel a jelenleg piacon kapható számítógépek már több maggal rendelkeznek. A beágyazott alkalmazásokhoz fejlesztett számítógépek is több processzormagot alkalmaznak. A többprocesszoros ütemezést több CPU-val rendelkező rendszerekben vagy több magos/szálas CPU-k esetében lehet használni. Az ütemezési algoritmusokat heterogén és homogén rendszerek csoportjára bonthatjuk. Heterogén rendszer esetében egy folyamat csak 1 CPU-n futhat. Homogén rendszer esetében az induló folyamat a rendszer közös sorába kerül. Homogén ütemezés esetében beszélhetünk aszimmetrikus és szimmetrikus rendszerről. Aszimmetrikus rendszer esetén egy közös (meghatározott CPU-n futó) ütemező van, a szimmetrikus esetében viszont minden CPU saját ütemezőt futtat.

6.2 Task-ok közötti kommunikáció

Mivel a rendszer működése közben a task-ok egymással párhuzamosan futnak, gondoskodni kell arról, hogy egyazon I/O-t, perifériát vagy memória területet két vagy több folyamat ne használjon egyszerre, mert abból hibás rendszerműködés alakulna ki. A taszkok közötti kommunikációra a következő módszerek állnak rendelkezésre:

- Szemafor (semaphore), mely 1 bit információ átadására alkalmas.
- Események (event flags), melyek több bit információ kicserélésére is alkalmasak.
- Postaláda (mailbox), amely komplexebb struktúra átadására szolgál.
- Sor (queue), amely több mailbox tömbjében tartalom átadására szolgál.

- Cső (pipe), amely direkt kommunikációt tesz lehetővé két taszk között.

A szemafor egy absztrakt adattípus, amelyet leginkább közös erőforrásokhoz való hozzáférés kontrollálására (kölcsönös kizárás) használnak. Ezen kívül alkalmas még egy esemény bekövetkeztének jelzésére, két folyamat tevékenységének összehangolására, és folyamatok szinkronizálására is. Szemafor típusai a következők lehetnek:

- Bináris szemafor (binary semaphore), amely egyszerű igaz-hamis jelzésre szolgál. Csak egyetlen erőforrás vezérlésére használható.
- Számláló szemafor (counting semaphore): A szemaforhoz egy számot rendelünk, működés közben a szemafor wait() művelete blokkol, ha a számláló 0 érékűre változik. Ellenkező esetben eggyel csökkenti a számláló értékét. A szemafor signal() művelete eggyel növeli a számlálót.
- Erőforrás szemafor (resource semaphore): Csak az a taszk engedhető el, amelyik lefoglalta az adott perifériát. Közös erőforrás védelmére jó, de taszkok közötti szinkronizációra nem alkalmas.
- Mutex: Egy bináris szemafor, mely kibővített tulajdonságokkal rendelkezik.

7 Szoftverfejlesztési szabványok

7.1 CMMI modell

A CMMI (Capability Maturity Model Integration) egy folyamatfejlesztési szemlélet, amely a hatékony folyamatok alapvető elemeit ismerteti meg a különböző szervezetekkel. Útmutatóként használható egy projekt, egy részleg, vagy akár egy teljes szervezet folyamatainak fejlesztésére. Segítséget nyújt a hagyományosan elkülönülő vállalati funkciók integrálásában, folyamatfejlesztési célokat és prioritásokat tűz ki, irányelveket ad a minőségügyi folyamatokhoz, valamint vonatkoztatási pontként szolgál a meglévő folyamatok értékeléséhez. A CMMI modellt a Carnegie Mellon Egyetem Software Engineering Institute-ja fejlesztette ki, az Amerikai Védelmi Minisztérium támogatásával.

A modell kidolgozásának célja a korábban a szoftverfejlesztésben, rendszerfejlesztésben és termékfejlesztésben leggyakrabban alkalmazott modellek, megközelítések összevonása volt egyetlen modellé, melyet bármely, szoftverfejlesztéssel foglalkozó szervezet alkalmazhat a teljes szervezet érettségének és / vagy egyes folyamatai képességének növelésére.

A fentiekből következik, hogy a CMMI modell a következő területeken (diszciplínákban) alkalmazható:

- szoftverfejlesztés (SW),
- rendszerszervezés- és fejlesztés (SE),
- integrált termék- és folyamatfejlesztés (IPPD).

A szoftverfejlesztés a CMMI modell kötelezően választandó diszciplínája. A szervezet tevékenységének típusa szerint ez a diszciplína kiegészíthető a rendszerszervezésre- és fejlesztésre vonatkozóval, és/vagy az integrált termék- és folyamatfejlesztésre vonatkozóval. A SE diszciplína nem tartalmaz további elemeket a SW diszciplínához viszonyítva: a modell ugyanazon követelményeit különbözőképpen kell értelmezni a szoftver- illetve a rendszerfejlesztés esetében. Az integrált termék- és folyamatfejlesztés egy sajátos működési mód a szervezeten belül: a rendszer/szoftverfejlesztés különálló csapatokban történik, s ezen tevékenységének összehangolása külön figyelmet, energiát igényel. Ha egy szervezet „integráltan” működik, bizonyos plusz követelményeket is ki kell elégítenie.

A diszciplína kiválasztása után a CMMI modell alkalmazásához a modell lépcsős vagy folytonos megközelítését is ki kell választani.

A lépcsős megközelítés a szervezet egészére vonatkozóan határoz meg érettségi szinteket. A folytonos megközelítés az egyes folyamatokra vonatkozóan képességi szinteket azonosít.

7.1.1 Folyamatközpontú szemlélet

A CMMI a folyamatok fejlesztéséhez nyújt segítséget. Folyamat alatt valamely kitűzött cél elérése érdekében végrehajtott tevékenységek halmazát értjük. A SEI kutatásai szerint a vállalatok számára a következő három szempont a legfontosabb az üzleti eredmények javításához: az emberek, a módszerek és az eszközök. Véleményük szerint épp a folyamat az, amely összefogja ezt a három szempontot. A folyamatot tekinthetjük a „ragasztó anyagnak”, amely összefogja az egyes tényezőket. Bárki egyértelműen felismerheti a fontosságát a

motivált, minőségi munkaerő meglétének, ez azonban mit sem ér, ha a folyamatot nem értik meg egyértelműen, vagy az nem működik megfelelően. Egy rendszer minőségét nagyban meghatározza a fejlesztésénél és karbantartásánál alkalmazott folyamatok minősége. Miért is éri meg fejleszteni a folyamatainkat? A számos előny közül a legfontosabbak talán a következők: csökkenő költségek, megnövelt hatékonyság, vevői elégedettség, jobb minőség, gyorsabb befektetési megtérülés, egyszerűbb költségbecslés és csökkenő életciklus idők.

Az utóbbi évtizedek elméleteit a gyakorlatban is kipróbálták, és a bevált megoldásokat felhasználva folyamatfejlesztési modelleket dolgoztak ki. A folyamatfejlesztési modell nem más, mint olyan elemek rendszerezett gyűjteménye, amelyek leírják a hatékony folyamatok jellegzetességeit. Ezeket a modelleket a következőkre használhatjuk:

- segítségükkel könnyebben kitűzhetőek a folyamatfejlesztési célok és prioritások;
- segítik biztosítani a stabil, kiforrott és megfelelő folyamatok létrejöttét;
- útmutatóként szolgálnak a projekt- és szervezeti folyamatok fejlesztéséhez;
- egy mérési módszer felhasználásával vizsgálható lesz a fejlesztési lépések állapota.

7.1.2 CMMI modellértelmezések

A CMMI terminológia szerint a modellértelmezés hasonlít az adatbázisoknál használt nézetekhez. Ugyanis minden egyes értelmezésnél ugyanazokat az adatokat vizsgáljuk, az egyedüli különbség azok szervezésében és megjelenítésében mutatkozik.

A CMMI kétféle megközelítést biztosít a szervezeteknek folyamataik fejlesztéséhez. Kiindulhatnak a folyamatterületeik képességeiből, de vizsgálhatók szervezeti fejlettség szempontjából is. A CMMI modellek mindkét szemléletmódot támogatják egy-egy modellértelmezési típussal.

Az értelmezések közötti választást az határozza meg, hogy a vállalat milyen szempontok, kritériumok alapján szeretné fejleszteni a folyamatait. Mindkét értelmezésnek megvannak az előnyei és a hátrányai is. A következő pontokban az egyes modellértelmezések jellegzetességeinek ismertetése található.

7.1.2.1 Folyamatos modellértelmezés

A folyamatos modellértelmezés a folyamatterületek képességeinek vizsgálatát helyezi előtérbe. Maximális rugalmasságot biztosít a szervezetek számára folyamataik fejlesztésében. Összpontosíthatnak akár egyetlen folyamathoz kapcsolódó problémás terület javítására, vagy akár a szervezet üzleti céljainak eléréséhez szükséges néhány területre is. Így fontossági sorrendet állíthatnak fel az egyes folyamatok között, csökkentve a kritikus területek kockázati tényezőjét. A folyamatterületek elkülönült fejlesztése miatt lehetőséget biztosít arra, hogy a szervezet különböző részeiben előforduló folyamatok teljesítményét összehasonlíthassák. Ennél az értelmezésnél nyilvánvalóan korlátozottak a vállalat választási lehetőségei, mivel számos olyan folyamat létezik, amelyeket nem lehet különválasztani, az egyik fejlesztése maga után vonja a másik fejlesztését is.

Azon szervezeteknek érdemes ezt a modellértelmezést választaniuk, amelyek tisztában vannak azzal, hogy mely folyamataikat kell fejleszteni, valamint átlátják a többi, a CMMI-ban definiált folyamatterülettel való összefüggéseket.

A folyamatos modellértelmezés úgynevezett képesség szinteket (capability levels) állít fel, amelyek kötött sorrendet írnak elő a folyamatok fejlesztéséhez. Öt darab szintet hoztak létre: CL1, CL2, CL3, CL4, CL5. Ezek a szintek egymásra építenek, így biztosítják a folyamatos fejlődés lehetőségét. A kötött sorrend lehetővé teszi, hogy nyomonkövessék, kiértékeljék és szemléltethessék egy szervezet folyamatainak fejlődését egy adott területen belül.

7.1.2.2 Lépcsős modellértelmezés

A lépcsős modellértelmezés szisztematikus, rendszerezett módon közelíti meg a modellközpontú folyamatfejlesztést. Ennél az értelmezésnél is különböző szintek vannak, azonban egyszerre csak egy szinttel léphet feljebb egy szervezet. Minden egyes elért szint tanúsítja, hogy megfelelő folyamatinfrastruktúrát hozott létre a vállalat ahhoz, hogy a következő szinthez tartozó követelmények kidolgozásába belekezdjen. Ez a minősítés lehetővé teszi, hogy a különböző szervezeteket összehasonlíthassák, hiszen egy egyszerű számmal jellemezhető egy adott szervezet fejlettsége.

Ennél a modellértelmezésnél úgynevezett fejlettségi szinteket (maturity levels) vezettek be. Összesen öt darab ilyen lépcső van: ML1, ML2, ML3, ML4, ML5. Ezek a szintek a kiválasztott folyamatterületeket öt fejlettségi lépcsőbe sorolják be, amelyek támogatják és segítik a szervezet folyamatainak jobbá tételét. Az egyes lépcsőket úgy alakították ki, hogy mindegyikük rendelkezzen alapfeltételként az alacsonyabb szintek követelményeivel. Egy adott szint elérése mutatja egy adott vállalat folyamatainak fejlettségét.

Ezt a fajta megközelítést azon szervezetek részesítik előnyben, amelyek nem tudják, hogy hol kezdjék el a fejlesztési folyamatot, illetve hogy mely folyamatokat fejlesszék. A lépcsős megközelítés a több évtizedes kutatások és tapasztalatok alapján minden egyes szinten kimondja, hogy mely folyamatokat milyen szinten kell javítani.

7.1.3 Fejlettségi szintek

A fejlettségi szint egy jól definiált lépcsőfok a szervezet folyamatfejlesztési tevékenységének jellemzésére. Minden egyes fejlettségi szint elérése a szervezet folyamatainak egy fontos részhalmazát javítja, felkészítve azokat arra, hogy a következő szintre léphessenek. Összesen öt darab fejlettségi szint van. Minden egyes lépcsőfok egy réteg az összefüggő folyamatfejlesztéshez. Az alapvető vezetési tevékenységekből kiindulva jól definiált és bizonyított úton lehet eljutni a magasabb szintekre. Mivel ezek a szintek szorosan egymásra épülnek, alapot képezve a következő szinteknek, ezért fontos megemlíteni, hogy a magasabb szintű folyamatokat kevesebb eséllyel lehet sikeresen végrehajtani, ha nem alapozzák meg őket az alacsonyabb szintek tapasztalataival.

Egy adott fejlettségi szint előre definiált folyamatok halmazához rendelt általános és különleges tevékenységek sorából áll, amelyek végrehajtása javítja a szervezet összesített teljesítményét. Egy adott szervezet elért szintje megjósolhatóvá teszi a szervezet teljesítményét egy bizonyos területen. Az egyes szintek elérését a kitűzött általános és különleges célok elérésének vizsgálatával lehet megállapítani.

5 A folyamatok fejlesztése kap hangsúlyt — Optimalizáló

4 A folyamatok mérésekkel alátámasztottak és szabályozottak — Mennyiségileg irányított

3 A folyamatok a szervezethez vannak igazítva és eredményesek — Definiált

2 A folyamatok projekthez vannak igazítva, gyakran eredménytelenek — Irányított

1 A folyamat ad-hoc, kaotikus jellegű, rosszul irányított , eredménytelen — Kezdeti

7.1. ábra A CMMI által definiált szintek

7.1.4 Folyamatterületek

Folyamatterület alatt egymáshoz kapcsolódó tevékenységek halmazát értik egy területen belül, amelyeket ha együttesen végrehajtanak, akkor kielégítenek számos olyan célkitűzést, amelyeket fontosnak tartanak az adott terület fejlődése érdekében. A CMMI v1.2 összesen 22 folyamatterületet tartalmaz. Ezek a területek közösek a folyamatos és a lépcsős modellértelmezésekben. A folyamatterület nem egy folyamat leírás, hanem tevékenységek halmaza. A különböző fejlettségi szintek ezen folyamatterületek részhalmazaihoz tűzik ki az általános és különleges célokat, amelyek eléréséhez definiálják az általános és különleges tevékenységeket.

A folyamatterületek leírásai a következő elemeket tartalmazzák:

- célkitűzés
- bevezető megjegyzések
- kapcsolódó folyamatterületek
- tevékenység-cél kapcsolatok táblázata
- különleges célok és tevékenységek
- általános célok és tevékenységek
- tipikus munkatermékek
- résztevékenységek
- megjegyzések
- szakterületekhez kapcsolódó kiterjesztések
- általános tevékenységek részletes kifejtése

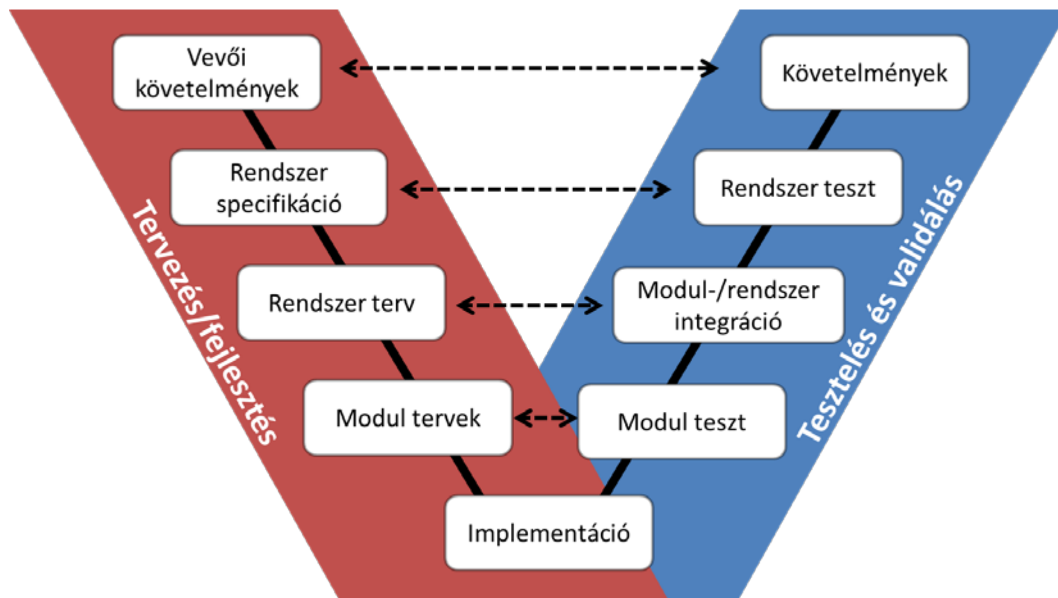
A következő táblázat felsorolja az egyes fejlettségi szintekhez tartozó folyamatterületeket.

Szint	Hangsúly	Folyamatterület
5 - Optimalizáló	Állandó folyamat fejlesztés	Szervezeti megújulás és fejlődés Kauzális analízis és feloldás
4 - Mennyiségileg irányított	Mennyiségi menedzsment	Szervezeti folyamat-teljesítmény Mennyiségi projekt menedzsment

3 - Definiált	Folyamat szabványosítás	Követelmények előállítása Technikai megoldás Termék integráció Ellenőrzés Validáció Szervezeti folyamat vizsgálat Szervezeti folyamatdefiniálás Szervezeti oktatás Integrált projekt menedzsment Rizikó menedzsment Döntés analízis és feloldás
2 - Irányított	Alapvető projekt menedzsment	Követelmény menedzsment Projekttervezés Projekt nyomkövetés és szabályozás Beszállítói megállapodás menedzsment Mérések és elemzések Folyamat- és termék minőségbiztosítás Konfiguráció kezelés
1 - Kezdeti		

7.2 V-modell

A V-modell a korai modellek családjába tartozik, melyet a német védelmi minisztérium fejlesztett ki és főleg a német hadsereg szoftverfejlesztéseiben vált használatossá. Az elnevezés nemcsak életciklus modellt, hanem egy teljes módszertant jelöl, aminek több elemét az ISO 12207 szabvány is átvette. A V-modell az egyes fázisok időbeli sorrendje mellett azt is definiálja, hogy az egyes fázisokban mely korábbi fázisok eredményeit kell használni; illetve az adott fázis tevékenységét és termékét mely korábbi fázisban leírt követelmények, illetve elkészített tervek alapján kell ellenőrizni. A V-modell használata főleg a biztonságkritikus számítógéprendszerek fejlesztése esetében a terjedt el.



7.2. ábra A V-modell

A 7.2. ábra a fejlesztés két folyamatának két megközelítését tükrözi. Top-down megközelítésként kifejezi a tervezési folyamat fentről lefelé történő haladását a baloldali ágban, míg a tesztelési folyamat lentről felfelé halad bottom-up megközelítésben, a jobboldali ágban. A gyakorlatban a projekttől és a szoftverterméktől függően a V-modellnek lehet több, kevesebb, vagy eltérő fejlesztési és tesztelési szintje. Például a komponensteszt után következhet komponens integrációs teszt vagy a rendszerteszt után rendszer integrációs teszt. A tervezés gyakran nagyszámú iterációt foglal magában, olyan műveletek sorával, amelyeket addig kell ismételni, amíg kielégítő eredményre nem jutunk. Fejlesztési fázisok a V-modellben:

- Követelmények specifikálása: a fejlesztési folyamat kiindulási pontját képező követelmények feltárása, elemzése, majd specifikációja történik. A fázis eredménye egy dokumentum, amely részletes információt tartalmaz a rendszer szolgáltatásairól és megszorításairól.
- Hazárdok és kockázatok elemzése: célja a lehetséges veszélyhelyzetek meghatározása a rendszerben, a megelőző kiszűrés érdekében. Az analízisek elvégzéséhez különféle módszerek állnak rendelkezésre. A kockázatok elemzésére valószínűség számítási segédeszközöket alkalmaznak. Az elemzési folyamatok eredményeként létrehozandó a biztonsági követelmények dokumentációja.
- Teljes rendszer-specifikáció: a funkcionális követelmények valamint a biztonsági követelmények együttese alkotja. Mindezen specifikáció alapján megkezdhető a teljes rendszer konkrét tervezési folyamata.
- Architekturális tervezés: a teljes informatikai rendszer hardveres és szoftveres architektúrájának megtervezése. A tervezésnek ebben a fázisában azt kell eldönteni, hogy mely funkciók legyenek megvalósítva hardver, és melyek szoftver által.
- A szoftver modulokra bontása: a fázisban a fejlesztési folyamatot további kisebb részekre, úgynevezett modulokra bontjuk fel a tervezési folyamat egyszerűsítése, áttekinthetőbbé tétele végett. A tervezés eredményeként a szoftver modulok specifikációja, valamint a köztük levő kapcsolódási folyamatok terve készül el.

- A modulok elkészítése és tesztelése: a szakaszban egyes modulok teljes implementációja valósul meg, ezután az elkészült modulok önálló tesztelése következik. Célszerű a tesztelési folyamatokat szintén előzetesen megtervezni.
- Rendszerintegráció: ebben a fázisban az elkészült szoftver-modulok integrálása történik egy teljes rendszerré, miután mindegyik modul átment a tesztelésen.
- Rendszerverifikáció: ezen fázis feladata annak az eldöntése, hogy rendszer megfelel-e a specifikációjának, funkcionálisan teljesíti-e az összes specifikációs pontot.
- Rendszervalidáció: el kell dönteni, hogy a teljes rendszer megfelel-e minden további, nem funkcionális követelmények. Ebbe beletartozik a biztonsági feltételek teljesítésének eldöntése is: az ún. biztonság-igazolás.
- Bizonylatolás (certification): a hatósági előírások és szabványok szerinti megfelelés eldöntése, és az erre vonatkozó bizonylatok kiállítása.
- A rendszer üzemeltetése: üzembe helyezés, üzemeltetés, karbantartás, elavulás, üzemeltetés megszüntetése.

8 MISRA-C

Míg a szabványos ISO C változatok specifikálása igen szabatos. Sok olyan szintaktikailag helyes, de szemantikailag rossz megoldást lehet létrehozni, amely biztonságkritikus alkalmazásokat használó iparágakban (pl. járműipar, repülőgépipar) nem megengedett. Ahhoz, hogy elkerülhessük ezeket a szintaktikailag helyes, de szemantikailag veszélyes kódrészleteket, a C nyelvi készletét korlátozni kell.

A MISRA C (Motor Industry Software Reliablility Association) egy szoftverfejlesztési szabvány, C programozási nyelvhez fejlesztve. Célja, hogy megkönnyítse a beágyazott rendszerekhez történő szoftverek esetében a biztonságos, hordozható és megbízható szoftverkód létrehozását.

A MISRA széles körben elfogadott „jó gyakorlat"-ként fejlődött ki a vezető iparágak által, beleértve a repülőgépipar, a távközlés, az orvosi eszközök, a vasút és járműipar résztvevőit.

A MISRA C szabvány első változata "A C nyelv használata gépjárművek szoftver fejlesztésében" címmel jött létre 1998-ban, ez hivatalosan MISRA-C:1998 néven ismert.

Célja az volt, hogy javítsa az UK (EgyesültKirályság) autóiparában használt szoftverek minőségét. A MISRA-C 1998 sikere nem csak az autóiparra terjedt ki, hanem széles körben alkalmazásra került a repülőgépiparban, valamit az egészségügyi iparban is. 2004-ben jött létre a második kiadás: "A C nyelv használata biztonságkritikus rendszerekben" (MISRA-C:2004) ez alapvető változásokat tartalmazott az eredeti szabványhoz képest.

A MISRA-C 2004 -es szabványverziója az, amely elfogadásra került az USA-ban (SAE J2632) és Japánban is. A MISRA-C 2004 javítja a MISRA-1998 hibáit, és pontosítja is azt. A szabvány 122 Mandatory (kötelező) és 20 Advisory (javasolt) szabályt tartalmaz a C nyelv leszűkítésére. A MISRA C röviden összefoglalva a következő célokat tűzte ki:

- Szintaktikailag helyes, szemantikailag rossz megoldásokra felhívni a figyelmet.
- Tiltani a nem egyértelmű változó típus használatot.
- Szabályozni a precedencia zárójelezéseket.
- Tiltani a nem strukturált programozást eredményező szerkezeteket.

A 2013. március 18-án bejelentették a MISRA C:2012 megjelenését, amely kiterjeszti támogatását a C nyelv C99-es változatára is (a C90 iránymutatásokat is tartalmazza), valamint számos olyan fejlesztést tartalmaz, amely csökkentheti az alkalmazás költségeit és komplexitását, míg segíti a C következetes, biztonságos használatát a biztonságkritikus rendszerekben. A MISRA C nem nyílt szabvány. Az útmutató dokumentumokat a felhasználóknak és a fejlesztőknek meg kell vásárolni. A MISRA közösség a C-n kívül még a C++ programozási nyelvhez is kidolgozott szabályrendszert.

8.1 Hogy néznek ki a MISRA szabályok?

A MISRA-C minden szabályhoz társít szabályszámot, kategóriát, követelményszöveget és forrás referenciát. Megadja a szabály célját, és általában egy példával illusztrálja a nem megfelelő kódot. A következő szabály a MISRA-C: 2004 9. szakaszából való: biztosítja, hogy minden automatikus változó kapjon értéket használat előtt:

Szabály 9.1 (Kötelező)

Minden automatikus változónak értéket kell kapnia használat előtt.

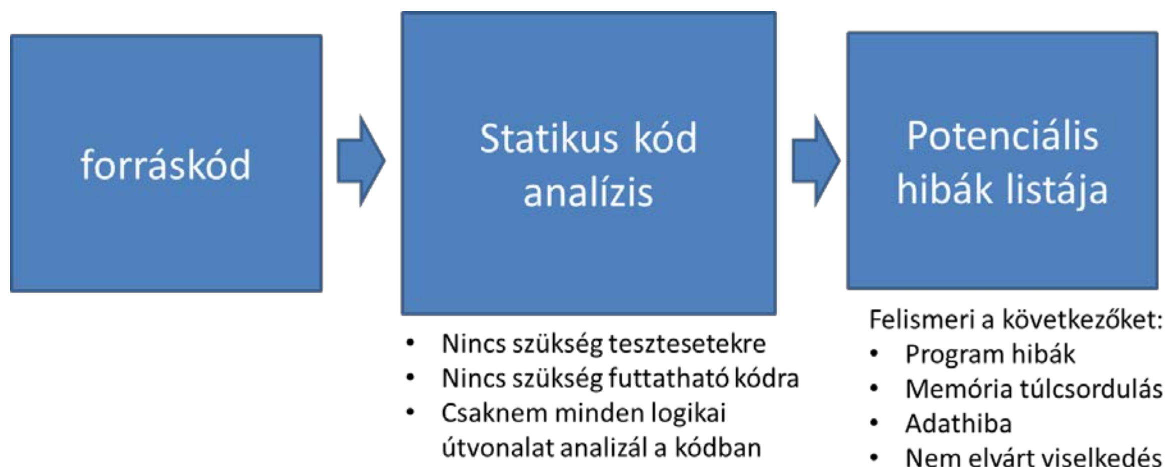
Magyarázat – Azok a változók adatfolyam anomáliákat okoznak, melyeket értékadás előtt használnak.

```
{
      int x;
      int y;
      x = y;
}
```

8.2 Statikus kódelemzés

A hibák felderítésének nagyon hasznos módja a programok vizsgálta: problémák feltárása a rendszer statikus reprezentációjának analízise segítségével (statikus verifikáció). A statikus analizátorok a forráskódok feldolgozására szolgáló szoftver eszközök, melyek a programkód elemzésével potenciális hibalehetőségek felfedezésére szolgálnak.

A MISRA szabály összeállítása rendkívül értékes. Ha nem lenne olyan megoldás, amely alkalmas arra, hogy automatikusan alkalmazza a szabályokat az elkészített kódra, nagyon kis hatással lenne a szoftver minőségére. A MISRA-C nagy hangsúlyt helyez a statikus kódelemző eszközök használatára. A szabályok nagy többségét nagyon nehéz lenne manuálisan ellenőrizni, mivel a rendszerkomponensek közötti adatfolyamot és a használt háttérváltozók tulajdonságait vizsgálják. A szabályrendszer néhány része a szükségtelen, nem biztonságos, vélt vagy nem pontos típuskonverziók megelőzését célozza.



8.1. ábra A statikus kódelemző célja

A statikus analízis felhasználása:

- Nagyon hasznos olyan nyelveknél, ahol a típusellenőrzés gyenge, és így a fordító sok hibát nem tud észlelni (pl. C).
- Kevésbé hasznos erős típusellenőrzéssel ellátott nyelvek esetén, ahol sok hiba fordítás közben kiderül (pl. Java).

A statikus analízis lépései:

- Vezérlés analízis. Hurkok többszörös belépési vagy kilépési pontokkal, nem elérhető kód, stb.
- Adathasználat analízis. Nem inicializált változók, többször írt változók közbülső értékadás nélkül, deklarált, de nem használt változók, stb.
- Interfész analízis. Konzisztens eljárás-deklaráció és használat.
- Információfolyam analízis. A kimenő változók függőségeinek feltárása. Önmagában nem tud anomáliákat detektálni, de kijelöl kódrészleteket a vizsgálat céljára.
- Útvonal analízis. Útvonalakat keres a program végrehajtása során és felsorolja a végrehajtott utasításokat.

9 Tesztfeladatok

Mit jelent a „code-review” a szoftver-fejlesztési folyamatban?

Mi a MISRA-C2 és miért hasznos?

Mi az FPGA?

Mi a V-model?

Mutassa be a Round-Robin algoritmust részletesen (előnyei, hátránya, folyamata)?

Mitől lesz jó egy beágyazott rendszer? Milyen rendszereket nevezünk beágyazott rendszereknek? (Adjon példákat!)

Vázolja fel a Neumann-architektúrát és adja meg jellemzőit!

Vázolja fel a Harvard-architektúrát és adja meg jellemzőit!

Hogy szól a lehetetlenségi tétel (Two General’s Problem, Szövetség a völgy felett)?

Adja meg egy Task állapotait az ütemezés szempontjából!

10 Irodalomjegyzék

Jones, Nigel. "Arrays of Pointers to Functions," Embedded Systems Programming, May 1999, p. 46.

Jones, Nigel. "Beginners Corner: Lint," Embedded Systems Programming, May 2002, p. 55.

ROBERT BOSCH GmbH (1991). *CAN Specification 2.0.* Robert Bosch GmbH, Stuttgart

dr. Dabóczi Tamás: Beágyazott rendszerek, Budapesti Műszaki és Gazdaságtudományi Egyetem, Méréstechnika és Információs Rendszerek Tanszék (2007)

Nicolas Navet, Francoise Simont-Lion: Automotive embedded systems handbook, Industrial information technology series, CRC Press (2009)

Ficsor Lajos, Krizsán Zoltán, Dr. Mileff Péter: Szoftverfejlesztés, Miskolci Egyetem, Általános Informatikai Tanszék, 2011

John H. Davies: MSP430 Microcontroller Basics, Elsevier , 2008, ISBN: 978-0-7506-8276-3

Dimitrios Hristu-Varsakelis, William S. Levine: Handbook of Networked and Embedded Control Systems, Birkhauser, 2005, ISBN-10 0-8176-3239-5