

Nagyteljesítményű vészhelyzeti áramforrás tervezése

Release 0.35

DRAFT

Tartalomjegyzék

Köszönetnyilvánítás	5
Bevezetés	7
1. A DC/DC konverter felépítése	9
2. Az ARM processzorok	13
2.1. Történeti áttekintés	14
2.2. Az ARM Cortex-M3 felépítése	16
2.3. Szoftverfejlesztés ARM processzorra	19
2.3.1. C/C++ fordító fordítása ARM architektúrára	19
2.3.2. C/C++ program írása ARM architektúrára	21
2.3.3. Kód letöltése, hibamentesítés	27
2.4. Gyártóspecifikus hardver kezelése	36
3. Soros-CAN átalakító	43
3.1. Hardver kialakítása	43
3.1.1. Blokkvázlat, kapcsolási rajz	43
3.1.2. Az RS232 rövid áttekintése	45
3.1.3. A CAN busz működése	45
3.2. Szoftver megoldások	48
3.2.1. Üzenetsorok (queue-k)	48
3.2.2. Az STM32 soros (RS-232) interfészének használata	53
3.2.3. A CAN busz interfész használata	56
3.2.4. A megszakítási rutinok működése	62
3.2.5. A főprogram funkciói	70
3.2.6. Kommunikáció a PC-vel: CLI és GUI	78
4. A DC/DC konverter kialakítása	85
4.1. A DC/DC konverter felépítése	85
4.1.1. Blokkvázlat, működés	85
4.1.2. Kapcsolási rajza	85
4.1.3. A transzformátor méretezése	87
4.1.4. A szuperkapacitás méretezése, áramhatárolás	93
4.2. Szoftver keretrendszer	95
4.2.1. Az A/D átalakító használata	95
4.2.2. A PWM jel előállítás	97
4.2.3. A szabályozó algoritmus kiválasztása	99
4.2.4. A főprogram működése	100
4.3. Kommunikáció megvalósítása	100
4.3.1. Fizikai réteg	100
4.3.2. Adatkapcsolati réteg	100
4.3.3. Hálózati réteg	101
4.3.4. Szállítási réteg	101
4.3.5. Alkalmazási réteg	103
4.3.6. A kommunikációs protokoll megvalósítása	103
5. Összefoglalás, végkövetkeztetés	104

6. Felhasznált szoftverek	106
7. Felhasznált irodalom	106
8. Készítendő képek - Zita	106
9. Frissítési eljárás	107

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani

- **Dr. Schuster György** főiskolai tanárnak, a témavezetőmnek
- **Krüpl Zsolt** okleveles villamosmérnöknek, a konzulensemnek
- **Körmendi Zita** kommunikációs szakembernek, a lektoromnak
- **Kollár Zsolt** okleveles villamosmérnöknek, aki nagyon sokat segített a hibák kijavításában,
- az **ST Microelectronics Company**nek, mely számos mikrovezérlővel járult hozzá a diplomamunkám létrejöttéhez.

Bevezetés

Ebben a dolgozatban egy nagy teljesítményű DC/DC konverter kialakításáról fogok írni. Maga a DC/DC konverter egy kapcsoló üzemű, push-pull felépítésű tápegység, amely mellett, hogy a bejövő 240-300 V-os DC feszültségből 300 V DC-t állít elő a kimenetén, távolról irányítható is egy felhasználó program, vagy más vezérlő segítségével. Az elkészítendő berendezés egy vészhelyzeti tápenergia-ellátás része lesz, így gondosan kell megválasztani az áramkör részegységeit és a kommunikációs buszt.

A kommunikáció megvalósításához a CAN¹ buszt fogom használni, mert a beépített CAN vezérlő elvégzi a keretezést és bejövő üzenetek szűrését, prioritizálását. Emellett a CAN gondoskodik arról, hogy a kommunikáció megbízható legyen, vagyis amellett, hogy a magasabb prioritású üzenetek előnyt élveznek, a nem nyugtázott üzenetek újraküldését is elvégzi.

A DC/DC konverter és a soros-CAN átalakító ARM² Cortex-M3 processzor alapú mikrovezérlő köré épül. Azért választottam az ARM-ot, mert *de facto* szabvány a 32 bites beágyazott alkalmazások körében, kiváló a támogatottsága (C/C++ fordító, programozó szoftver, fórumok), számos gyártó ajánl ARM magos mikrovezérlőt (ST, Luminary, Atmel, stb.), és a Linux lefordítható ARM processzorra. Ez utóbbi azért fontos, mert a C/C++ fordító már kiforrottnak mondható amiatt, mert a Linux kernel fejlesztők jelezték az esetleges hibákat a fordító készítőinek. Az ARM Cortex-M3 mag használata mellett szól az az érv, hogy nagy teljesítményű (gyors), kifejezetten mikrovezérlőt alkalmazó beágyazott rendszerek számára fejlesztették ki. A lapkagyártók számos perifériával látták el, így jelenleg a technológia csúcsát képviselik. A CAN vezérlő is helyet kapott a perifériák között: természetesen magam is a mikrovezérlő beépített CAN controllerét fogom használni mind a soros-CAN átalakító megvalósításához, mind a DC/DC konverter megépítéséhez.

Aki ezt a leírást olvassa, remélem, hogy élvezettel teszi majd. Az olvasáshoz-tanuláshoz sok sikert és kitartást kíván:

A szerző

Budapest, 2008. augusztus 31.

1 Controllor Area Network - vezérlők helyi hálózata

2 Az ARM, a Cortex, a Thumb, az AMBA, az ABH, az APB és a CoreSight az ARM Limited bejegyzett márkanéve.

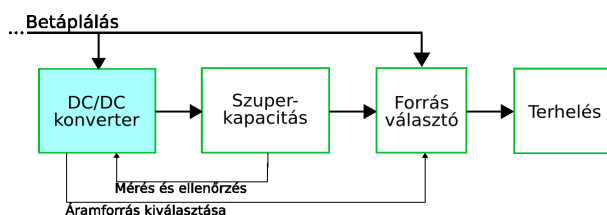
1. A DC/DC konverter felépítése

Ez a dolgozat azért született, hogy leírjam, miként terveztem és valósítottam meg a specifikációban meghatározott DC/DC konvertert és az ahhoz szorosan kapcsolódó ellenőrző-irányító rendszert.

Lássuk először a kiinduláshoz használt specifikációt, mely eredetileg az European Aeronautic Defence and Space Company EADS N.V.-től (Airbus gyártója) származik:

- A bemenő feszültség 240 V és 300 V között változhat ($270\text{ V} \pm 10\%$).
- A kimenő feszültség névleges értéke 300 V.
- Áramhatárolás értéke: 3 A.
- Maximális terhelhetőség: 1 kW.
- Szabályozó algoritmus: integráló típusú, véges beállítású.
- A bemenet és a kimenet galvanikusan nem elválasztott.
- Távvezérlés és ellenőrzés CAN buszon keresztül, felhasználói felület.

A vészhelyzeti áramforrás általános blokkvázlata a következő ábrán látható:



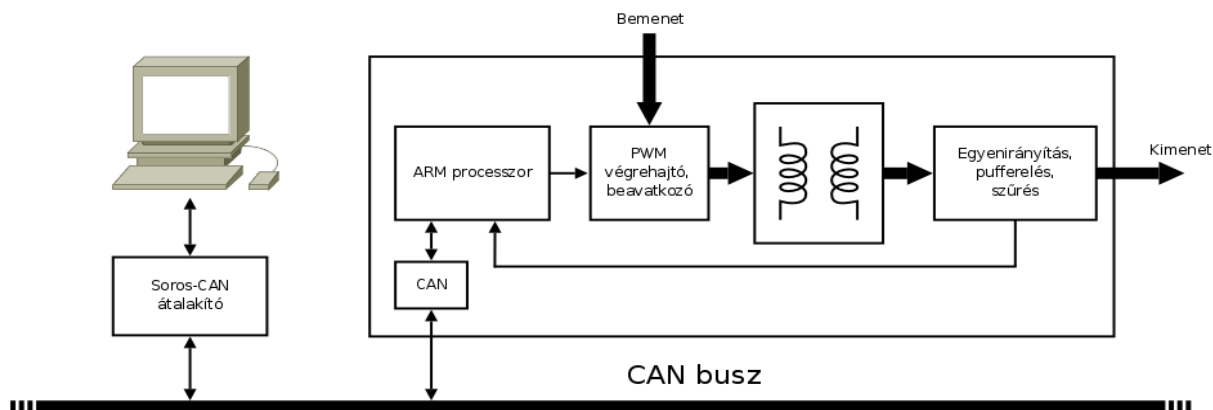
A „Betáplálás” vonalon érkező áram normál üzemben közvetlenül a terhelésre jut. Ha azonban ez az áramforrás valamilyen okból nem tud energiát szolgáltatni, szerepét a szuperkapacitás blokk veszi át. A két forrás közötti átkapcsolást a „Forrásválasztó” eszköz végzi, melyet a DC/DC konverterben található mikroprocesszor vezérel.

Normál üzemben a szuperkapacitás blokkot egy megfelelően felépített tápegység, mint feszültségghatárolt áramgenerátor tölti. A dolgozat fő témája ennek a DC/DC átalakítónak a megtervezése és megépítése.

* * *

A lehető legnagyobb hatásfok (tehát legkisebb veszteség) elérése érdekében kapcsoló üzemű megoldásban kell gondolkodnunk. A tényleges architektúra kiválasztása a 4.1.1. fejezetben történik.

A specifikáció alapján a következő felépítést célszerű használni:



Lássuk először, hogy a képen látható blokkok milyen feladatot látnak el:

A PWM³ elvű beavatkozó feladata, hogy a bemeneten érkező 240...300 V-os egyenáramú jelet váltakozófeszültséggé alakítsa. Ere azért van szükség, mert a transzformátor nem tud DC jelet átalakítani.

A transzformátor kimenetén akkora feszültség jelenik meg, amely még a legkisebb bemenő feszültség mellett is képes megfelelő szintű kimenő jelet szolgáltatni. A méretezés részleteit a 4.1.3. fejezet tartalmazza.

A váltakozó feszültség egyenárammá alakításáról az „Egyenirányítás, pufferekés, szűrés” blokk gondoskodik. A kapcsoló üzemi felépítés következményeként nagy mennyiségű rádiófrekvenciás zavar keletkezik (ennek maximális értékét szabvány írja elő). A zavarok szűrését is az előbb említett áramkör rész végzi.

A kimeneten mérhető feszültség és áram értékét az ARM alapú mikrovezérlő folyamatosan méri, figyeli. Ha kimenő áram eltér az előírt értéktől, akkor a mikrovezérlő beavatkozik oly módon, hogy a kimenő áram a meghatározott érték felé közeledjen (és lehetőleg érje is el azt). A kimenő áram szabályozása a feszültség változtatása által lehetséges.

A mikrovezérlő a szükséges szabályozási paramétereket CAN buszon keresztül kapja. A CAN lehetőséget ad arra is, hogy egy PC vagy más kijelző segítségével ellenőrizzük a szabályozási kört. Ehhez persze szükséges egy megfelelő felhasználói program is. Mivel a PC-k (és gyakran már elektronikus vezérlő áramkörök) nem rendelkeznek CAN illesztővel, ezért a a dolgozat első részében a soros-CAN átalakító megépítését ismertetem.

De miért pont a CAN buszra esett a választás? A kérdés jogos. Ha megnézzük a következő táblázatot, láthatjuk, hogy nagyon megbízható, de nagyobb távolságok áthidalására alkalmas megoldást kellett találni. Ebben az esetben a CAN látszik az optimális megoldásnak:

	RS-485	CAN	LIN	Ethernet
Huzalozás	sodrott érpár	sodrott érpár	aszimmetrikus	Sodrott érpár, full-duplex
Max. sebesség	10 Mbit/s	1 Mbit/s	19200 bit/s	100 ⁴ Mbit/s
Max. távolság	1000 m	1000 m	N × 10 m	100 m ⁵
Keretezés	szoftveres	hardveres	szoftveres	hardveres
Eszközök száma	32	32	16	kb. 100
Adatok keretenként	keretezéstől függ	0 - 8 bájt	2 - 8 bájt	46 - 1500 bájt
Magbízható átvitelt biztosít-e	nem	igen	nem	nem

3 Pulse Width Modulation – pulzusszélesség moduláció.

4 Mikrovezérlőkben megvalósítva 100 Mbit/s, de léteznek 1Gbit/s, sőt 10Gbit/s sebességű hálózatok is.

5 Aktív eszköz (pl. switch) nélkül.

	RS-485	CAN	LIN	Ethernet
Real-time megoldásban használható-e	igen	igen	igen	bizonyos megkötésekkel

Mivel a DC/DC konverter vezérléséhez és ellenőrzéséhez egy megbízható, és gyors busz szükséges, ezért a konverter megvalósításához a CAN-t fogom használni. Az STM32 csip rendelkezik beépített CAN vezérlővel, ezért ebből nem fog külön probléma adódni.

A PC-kben nem található CAN busz illesztő, ezért egy „külső” áramkörre, egy soros-CAN (RS232-CAN) átalakítóra is szükség lesz. A dolgozat második része ennek az átalakítónak a megépítésével foglalkozik.

Mind a soros-CAN átalakító, mind a DC/DC konverter ARM Cortex-M3-at tartalmaz, ezért a következő fejezet feladata, hogy összefoglalja mindazt, amit az ARM processzorokról tudni érdemes. Megismerhetjük az ARM processzorok történetét, általános felépítését, a különböző gyártók által beépített perifériákat. Természetesen meg kell néznünk a programozás módját is, láthatjuk, hogy hogyan lehet C fordítót fordítani az ARM processzorokhoz, az elkészített kódot hogyan tudjuk lefordítani és letölteni a mikrovezérlő nem felejtő (FLASH) memóriájában, és milyen lehetőségei vannak a szoftver könyvtáraknak (firmware library).

Az ezt követő fejezetben megismerkedhetünk a soros-CAN átalakító építésével, a gyártóspecifikus hardver programozásával, a CAN illesztő használatával. Egy Python program segítségével le is tesztelhetjük az elkészült áramkört.

Ezután következik a DC/DC konverter tervezése: az architektúra kiválasztása, a transzformátor méretezése, az egyenirányítás és a szűrés megválasztása, a rádiófrekvenciás zavarok elleni védelem. Nem mellékes az sem, hogy milyen szabályozó algoritmust használunk a kimenő feszültség névleges értéken tartásához. Mivel a DC/DC konverter nem egy individuális jellegű áramkör, ezért definiálni kell a kommunikációs protokoll részleteit, és egy megfelelő felhasználói programmal kell segíteni a felhasználó munkáját.

2. Az ARM processzorok

Felmerül a kérdés, hogy miért éppen ARM alapú mikrovezérlővel oldottam meg a kitűzött feladatot. A válasz alapvetően egyszerű: ez a processzor család tagjai kiváló tulajdonságokkal rendelkeznek, ezen tulajdonságok nagy mértékben lerövidítik és megkönnyítik a fejlesztést és tesztelést. Az általam ismert 8 bites mikrovezérlők egyike sem rendelkezik ilyen mértékű rugalmassággal, és teljesítményben, perifériakiépítettségben is elmaradnak a ma kapható ARM Cortex-M3 mikrovezérlőkhöz képest.

Az előbb említett tulajdonságok a következők (csak a legfontosabbakat emeltem ki):

- 32 bites felépítés. Ez lehetővé teszi, hogy az aritmetikai műveletek argumentumai 32 bitesek legyenek, ami nagyobb pontosságot tesz lehetővé.
- A 32 bites regisztereknek köszönhetően 4 GB címtartomány címezhető meg közvetlenül. Ebben a címtartományban vannak kialakítva a memóriák (FLASH és SRAM) és a perifériakészlet regiszterei is.
- 32 bites ARM és 16 bites Thumb utasítások (a Cortex-M3 csak a Thumb2-t ismeri) végrehajtására is képes.
- A processzor több futtatási módot tartalmaz: lehetőség van a rendszer- és a felhasználói kód szétválasztására. Ez növeli a beágyazott rendszer biztonságát és megbízhatóságát. Ezt persze nem feltétlenül szükséges igénybe venni.
- Számos egységet beépítve tartalmaz: gyakran MMU⁶-t, MPU⁷-t, FPU⁸-t, ETM⁹-et, megszakításkezelőt terveznek bele az ARM cég mérnökei.
- A csipgyártók még számos perifériával egészítik ki a processzor magot: számlálókkal, kommunikációs eszközökkel, (gyakran 12 bites) A/D és D/A átalakítókkal, stb.
- Kiváló szoftveres támogatással rendelkeznek: a GCC fordít ARM processzorra. Az OpenOCD¹⁰ számos csipgyártó termékét támogatja.

Az ARM család az idők folyamán számos taggal bővült. Az újabb tagok megjelenése nem mindig járt a processzor magjának teljes lecserélésével. Elmondható, hogy az ARM processzorok csoportosíthatók aszerint, hogy a processzor belseje milyen felépítéssel (architektúrával) rendelkezik. Ebből persze az következik, hogy egy adott architektúrát több processzorban is megtaláljuk.

Ha két processzor kialakításához ugyanazt az architektúrát használták, akkor vajon mi lehet a különbség a processzorok között? Úgy foglalthatnám össze, hogy a két mag ugyan megegyezik, de a processzorok más-más kiegészítővel rendelkeznek. Pl. az ARM966 nem tartalmaz MMU-t, míg az ARM926-ban ez ki van alakítva, pedig mindkét processzor architektúrája ARMv5. A csipgyártó szabadon eldöntheti, hogy szeretne-e MMU-t kialakítani a csipben. Ha igen, akkor az ARM926 mellett dönt, ha nincs szükség MMU-ra, akkor pedig az ARM966-ot vásárolja meg az ARM cégtől (az ARM cég nem gyárt csipet, csak megtervezi azt, és a kész terveket – intellektuális tulajdont (IP) – adja el).

Hogy kis rendet hozzak a káoszba, a következő részben összefoglalom, hogy a kezdeti időktől napjainkig hogyan alakultak az ARM processzorok.

6 Memory Management Unit – memóriakezelő egység, operációs rendszerek futtatásához szükséges.

7 Memory Protection Unit – memóriavédelmi egység, az egyszerűbb beágyazott rendszerek megvédhetik a memória egy részét az illegális hozzáféréstől.

8 Floating Point Unit – lebegőpontos egység, a lebegőpontos számok kezelésének hardveres támogatásához

9 Embedded Trace Macrocell – beágyazott nyomkövető egység, hibakeresésre használatos

10 Open On-Chip Debugger – nyílt forráskódú programozó és nyomkövető szoftver

2.1. Történeti áttekintés

Ebben a részben az ARM¹¹ processzorok fejlődésének menetét foglalom össze. Sajnos nincsen lehetőség arra, hogy minden processzor kiadásról részletesen írjak, így csak a jelentősebb állomásokat fogom szemügyre venni.

Az ARM (Advanced RISC Machine – fejlett, csökkentett utasításkészletű gép) processzorok fejlesztése 1983-ban kezdődött az Acorn¹² cégnél. Az első, ténylegesen használható kiadás az **ARM2**-es volt, és 1984-től érhető el a piacon. Ennek még 26 bites volt a PC¹³-je, és mindössze 30.000 tranzisztort tartalmazott (a Motorola 68000¹⁴-es processzora 70.000-et).

Ezután hihetetlen sebességgel folytatódott a fejlesztés. A következő, igen elterjedt processzor az **ARM7TDMI** volt (ARMv4-es architektúrával). Ma is számos gyártó alkalmazza mikrovezérlőkben:

- Atmel AT91SAM7¹⁵,
- NXP LPC2000,
- ST STR7,
- Analog Devices ADUC7000,
- Texas Instruments TMS470, stb.

Az ARM7TDMI rendkívüli népszerűségének alighanem az az oka, hogy a processzor elég egyszerű felépítésű, sem MMU-t, sem MPU-t, sem FPU-t, sem cache-t nem tartalmaz, így nagyon költséghatékonyan (olcsón) tudják a lapkagyártók előállítani. Egyszerűsége ellenére elmondható róla, hogy tartalmaz mindent, amit egy modern processzortól elvárunk: 32 bites regiszterek, processzor üzemmódok, kivételkezelés, 1 órajeles utasítások (ez csak részben igaz).

A fejlesztés hamar eljutott az **ARM9**-es processzorokhoz. Az ARM9-es processzorok ARMv4-es és ARMv5-ös architektúrájuk lehetnek. A régebbi processzorok (ARM9TDMI, ARM920, ARM922, ARM940) v4-esek voltak. Érdekes megfigyelni, hogy a régebbi architektúra ellenére az ARM cég mérnökei terveztek MMU-t és cache-t a processzorokhoz (ARM920, ARM922). Ez azért lehet érdekes, mert a Linux igényli az MMU meglétét, tehát ha egy processzor tartalmaz MMU-t, akkor azon képes lehet elfutni. És valóban, az ATMEL cég AT91SAM9200-as processzora köré épített demó boardon fel tud BOOT-olni a Linux.

A későbbi ARM9-ek már kivétel nélkül ARMv5 architektúrával készültek. Ezek nagy előnye, hogy képesek DSP¹⁶ utasítások futtatására (ha a processzor neve „E” betűt tartalmaz: mindegyik tartalmaz). Ha a processzor nevében „J” betű is található (ARM926EJ-S), akkor az képed Java bájtkódot natív módon futtatni. Ez mobil telefonok esetén lehet érdekes, ahol különösen fontos szempont a játékprogramok hordozhatósága.

Az ARM926EJ-S processzor különleges tulajdonsága, hogy rendelkezik MMU-val, és cache-sel, így építhető olyan áramkör, amin fut a Linux. Nürnbergben, a beágyazott rendszerek kiállításán nagyon sok olyan megoldást láttunk, amin valóban Linux futott. Az ATMEL cég kiváló processzorokat gyárt AT91SAM9260, AT91SAM9261, AT91SAM9262 és AT91SAM9263 néven.

A többi ARM9-es, és ARMv5-ös architektúrájú processzor nem tartalmaz cache-t, sőt gyakran még MPU-t sem. Ez a tulajdonságuk alkalmassá teszik ezeket az eszközöket, hogy mikrovezérlők processzorai legyenek. Ilyen lapkákat gyártott pl. az ST Microelectronics (STR9xxx ARM966E maggal – honlapjuk szerint elavult termék, gyártását befejezték).

¹¹ http://en.wikipedia.org/wiki/ARM_architecture

¹² http://en.wikipedia.org/wiki/Acorn_computers

¹³ Program Counter – utasításszámláló, az aktuális/következő utasítás címét tartalmazza.

¹⁴ http://en.wikipedia.org/wiki/Motorola_68000

¹⁵ <http://en.wikipedia.org/wiki/AT91SAM>

¹⁶ Digital Signal Processor – digitális jelfeldolgozó processzor

A mai, modern ARM-ok az **ARM11**-nél kezdődnek. Ezek valódi applikációs processzorok, arra tervezve, hogy PDA-k, palmtop-ok, GPS-ek, vagy bármilyen, számítás- és erőforrásigényes alkalmazások processzorai legyenek. Természetesen ezek mind képesek DSP utasítások futtatására, SIMD¹⁷ utasításokat is végrehajtanak, Java kódot is tudnak futtatni, az MMU mindegyikben megtalálható, opcionálisan lebegőpontos segédprocesszorral is fel vannak szerelve. Nem kisebb gyártók használják, mint a Nokia a mobiltelefonjaiban, az Apple az iPhone¹⁸-ban és az iPod touch-ban. Az ARM11 már ARMv6 architektúrájú.

Itt érkeztünk el napjaink legújabb processzoraihoz: az **ARM Cortex**-hez. Ennek az ágnak az ARM cég nevet adott, ezzel jelezve hogy fontos mérföldkőhöz érkeztünk. Ezek a processzorok (az M1-től eltekintve) ARMv7 architektúrájúak.

Az előbb említett **ARM Cortex-M1** ARMv6-os, és az az érdekessége, hogy FPGA-ban kiválóan használható szoftver processzorként. Méretére jellemző, hogy befér egy XILINX Spartan-3 FPGA-ba. Csak az új generációs Thumb2¹⁹ utasításokat tudja futtatni. Sem MMU-t, sem MPU-t, sem cache-t nem tartalmaz.

Az **ARM Cortex-M3** processzor már ARMv7-es, tehát a legújabb fejlesztéseket tartalmazza. Kifejezetten arra optimalizálták, hogy mikrovezérlő készüljön belőle. Csak Thumb2 utasításokat tud futtatni, azt viszont nagyon hatékonyan teszi. Opcionálisan MPU-val „felszerelve” szállítják. Jelenleg két csipgyártó forgalmaz ARM Cortex-M3 alapú eszközöket. Mindkettő beépíti az MPU-t. Itt mondanám el, hogy az ATMEL, a nagy múlttal rendelkező Zilog és az NXP is vásárolt ARM Cortex-M3 licenst, így a közeljövőben várható, hogy ők is elkezdnek ilyen csipeket forgalmazni.

Említettem, hogy az ARM cég néven nevezi ezt az ágat. Ez nem véletlen: párhuzam vonható a Cortex processzorok és a „klasszikus” csipek között. Tudását, komplexitását tekintve a Cortex-M3 leginkább az ARM7(TDMI)-tel mérhető össze.

Az **ARM Cortex-R4(f)** processzor leginkább az ARM9-hez hasonlít. Sokat azonban nem lehet tudni róla, mert az ARM honlapján nem található meg a megfelelő felhasználói kézikönyv. Annyi azonban bizonyos – megkérdeztem tőlük személyesen – hogy egy nagyon fejlett megszakításvezérlővel építik egybe, amely megszakításvezérlő képes több processzor között elosztani a megszakítási kéréseket (statikusan programozható). Opcionálisan lebegőpontos egységet (FPU) tartalmaz, erre utal az „f” betű a nevében. A Thumb2 mellett az ARM utasításokat is tudja futtatni.

A technika csúcsát az **ARM Cortex-A8** és az **ARM Cortex-A9** processzor képviseli. Képességeit tekintve az ARM11-hez állnak a legközelebb, de messze túl is szárnyalják azt. A ThumbEE-nek (Thumb Execution Environment – Thumb végrehajtási környezet) köszönhetően javul számos szkriptnyelvű kód (Python, Perl, Limbo, Java, C#) futtatási sebessége. A TrustZone (megbízható övezet) technológia azzal a képességgel ruházza fel a processzort, hogy képes legyen megbízható és kevésbé megbízható kódok egymástól teljesen szeparált futtatására. Ennek köszönhetően javul a rendszer megbízhatósága és biztonsága.

A csipgyártók közül csak a Texas Instruments használ ARM Cortex-A8 processzort. Ez mag található az OMAP3²⁰ (Open Multimedia Application Platform – nyílt multimédia platform) processzorban, ami kiválóan alkalmazható multimédiás termékekben (Pandora²¹).

Nem lehetetlen, hogy mire beadom ezt a dolgozatot, változni fog a helyzet, talán új processzorok jelennek meg, de egy valamit biztosan állíthatok: **az általam megépítendő DC/DC konverterben ARM Cortex-M3-at fogok használni.**

17 Single Instruction Multiple Data – egyazon utasítás több adaton

18 <http://en.wikipedia.org/wiki/iPhone>

19 16 vagy 32 bites utasítások, de nagymértékben eltérnek a „klasszikus” ARM utasításoktól, melyek mindig 32 bitesek.

20 <http://en.wikipedia.org/wiki/OMAP>

21 [http://en.wikipedia.org/wiki/Pandora_\(console\)](http://en.wikipedia.org/wiki/Pandora_(console))

2.2. Az ARM Cortex-M3 felépítése

A szoftverfejlesztéshez feltétlenül szükségesnek tartom, hogy tisztában legyünk az ARM processzorok felépítésével. Így kiderül majd, hogy miért jelent hihetetlen előnyt a lineáris címtartomány, vagy a perifériák adott memóriaterületre való „beépítése”.

ARM9-től kezdve a processzorok Harvard architektúrájúak. Nincsen ez másként a Cortex processzorok esetén sem. A Harvard architektúra annyiban különbözik a von Neumann architektúrától, hogy külön buszt használ az utasítások és az adatok „kezelésére”, vagyis más útvonalon érkeznek a végrehajtandó utasítások a FLASH memóriából, és más útvonalon közlekednek az adatok a processzor és a RAM között.

Azért, hogy a processzor végül mégis lineáris címekkel tudjon dolgozni, egy busz-mátrixot terveztek a gyártók a processzorhoz.

Az utasítások „hagyományosan” a FLASH-ból érkeznek a processzorba, az adatterület pedig a RAM-ba kerül. Ha egy konstans sztringet szeretnénk küldeni a felhasználónak (például soros porton keresztül), akkor nem lenne túl célravezető, ha a konstans, tehát nem változó karaktersorozatot a RAM-ban tárolnánk. De nem lenne éppen optimális megoldás az sem, ha egy DSP művelethez használt szinusz táblát (ami megint csak konstans adatokat tartalmaz) a RAM-ba töltenénk.

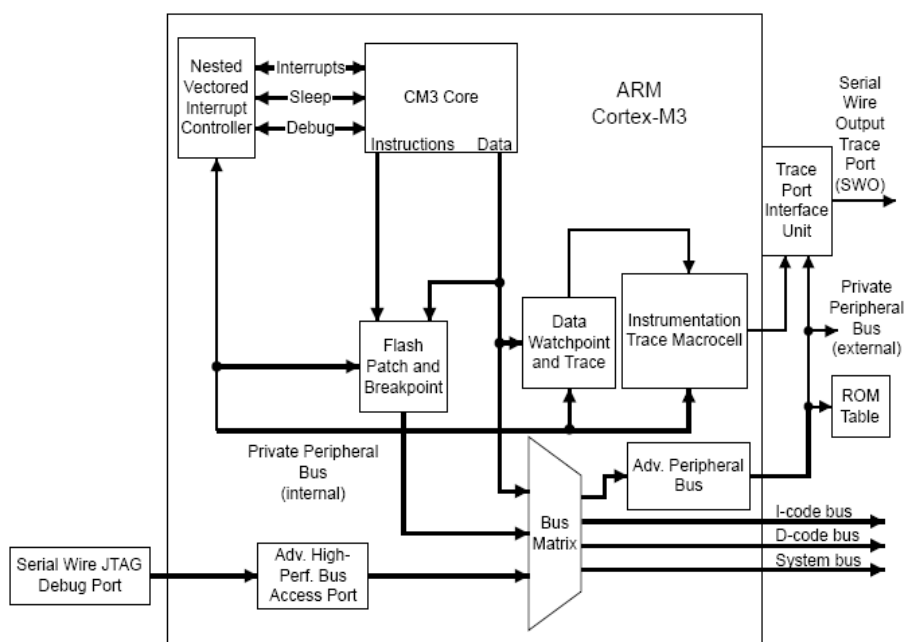
A megoldás ilyenkor az, hogy a FLASH-ben kijelölünk egy területet a konstans, csak olvasható (read only – .rodata) adatoknak, és amikor szükség van rájuk, akkor valamiképpen – a busz mátrixon keresztül – a FLASH-ból érjük el azokat.

Ugyan nem gyakori, de előfordulhat, hogy a programkódunk nem a FLASH-ben található, hanem a RAM-ban. Ez akkor célszerű, ha a végrehajtandó utasításokat dinamikusan kapjuk, például Etherneten keresztül. Ha ilyenkor nem akarjuk a kódot a FLASH-be égetni, tölthetjük egy szabad RAM-területre is. De – megint a busz-mátrix segítségével – gondoskodunk kell arról, hogy a RAM-ban levő kódot egyszerűen, lineáris címzéssel elérhessük.

A jó hír az, hogy a busz-mátrix mindezt (mármint az automatikus útvonal-kiválasztást) teljesen automatikusan elvégzi, így végül is nekünk nincsen tudomásunk arról, hogy egy szó (legyen az utasítás vagy adat) végül is melyik buszon jelenik meg.

Itt kell megemlíteni, hogy a perifériák is külön buszon csatlakoznak a rendszerhez (ábrán: APB), ennek kezelése is a busz-mátrix feladata.

Figure 2-1. CPU Block Diagram



Ezek után talán érthető, hogy a Harvard architektúra miért nem jelent hátrányt a mikrovezérlő kialakításakor. Sőt, tulajdonképpen profitálunk is a különválasztott buszokból: amíg egy utasítás eredményét menti a processzor (az adatbuszon (D-code bus) keresztül), addig a következő utasítás kódja már érkezik az utasításbuszon (I-code bus) át.

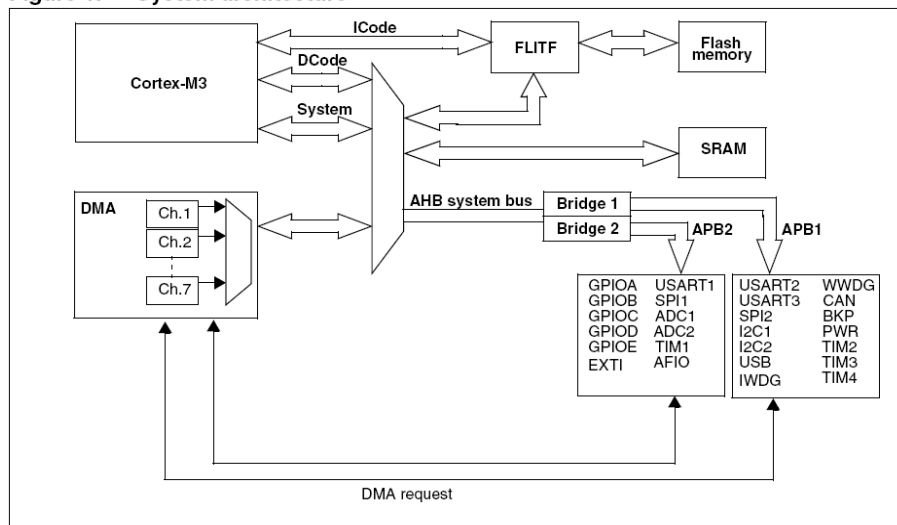
A lineáris címtartomány előnye akkor mutatkozik meg, mikor C nyelven programozunk. Az assembly utasításokat és a regiszterek közvetlen használatát megpróbálom minden erőmmel kerülni a diplomamunkám elkészítése során. Megtehetem azt, mert a C nyelv szinte minden hardverspecifikus dolgot elrejt előlem. Nincsen ez másként a memóriacímekkel kapcsolatban sem. C nyelvben a memóriacímeket ponttereknek, mutatóknak nevezik. Mivel a memóriacímek kivétel nélkül 32 bites számok, és a busz-mátrix teszi a dolgát anélkül, hogy nekem külön kellene foglalkoznom vele, ezért nem érdekes túlságosan az sem, hogy egy pointer hova is mutat, konkrétan: a 4 Gbájtos memóriatartomány melyik részét címzi.

Fontos része a Cortex-M3 magnak az NVIC, vagyis Nested Vectored Interrupt Controller²². Ez egy nagyon fejlett megszakításkezelő, lehetővé teszi a megszakítások prioritásának beállítását. Ez azért hasznos, mert a magasabb prioritású megszakítások megszakíthatják az alacsonyabb prioritású társaikat.

Az ábrán nem látszik, de a Cortex-M3 magnak része egy olyan számláló-időzítő, ami megszakításokat képes generálni, ezzel lehetővé teszi, hogy az operációs rendszer elragadja a vezérlést az aktuálisan futó alkalmazástól, és egy másik alkalmazásnak adja, így biztosítani tudja a feladatok látszólagos egymás melletti (egyidejű) futását. Ezt a képességet az idegenek multitaskingnak (több-feladatos működésnek) nevezik.

A processzor magjának áttekintése után vessünk egy pillantást a gyártók által hozzáadott perifériákra. Először álljon itt az általam nagyra tartott ST cég termékének blokkvázlata:

Figure 1. System architecture



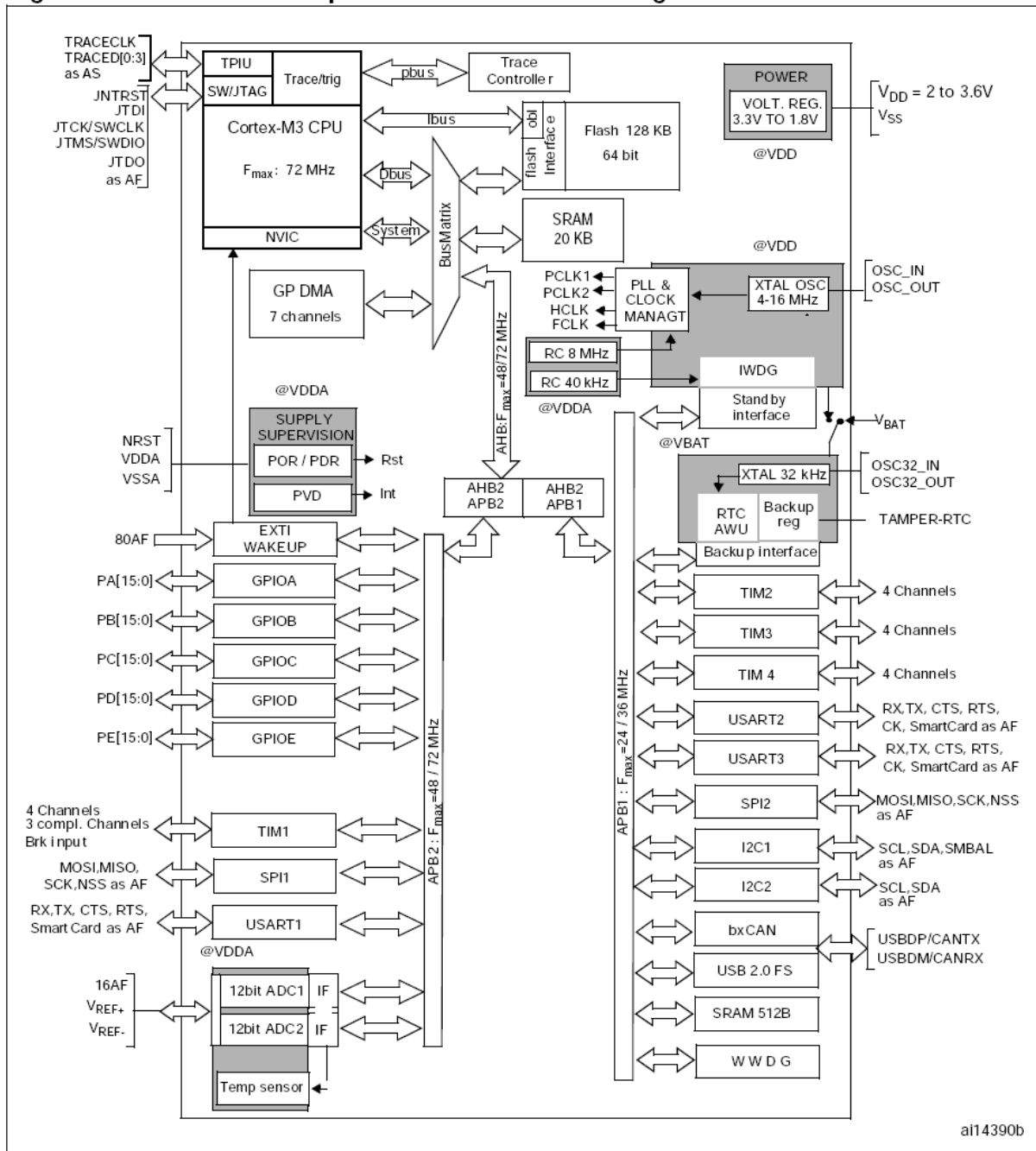
Érdekessége a lapkának a kialakított DMA, vagyis a direkt memória hozzáférést biztosító egység. Ez tehermentesíti a processzort azáltal, hogy az adatokat nagyobb blokkokban automatikusan másolja a memória és a perifériák között.

Látható az ábrán, hogy perifériában nincsen hiány: számtalan GPIO²³, USART, (12 bites) ADC, CAN, számlálók, stb. segíti a munkánkat. Egy jobb blokkvázlatot láthatunk a következő képen:

²² Nested Vectored Interrupt Controller – olyan megszakításvezérlő, amely lehetővé teszi a megszakítások egymásba ágyazását.

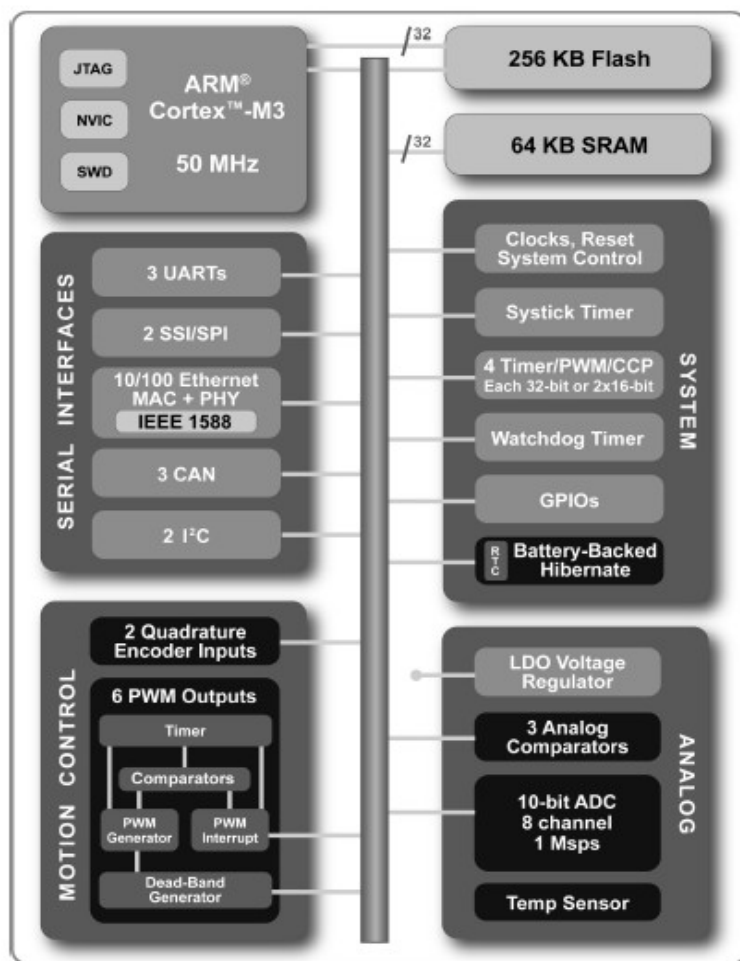
²³ General Purpose Input / Output: általános célú bemenet / kimenet

Figure 1. STM32F103xx performance line block diagram



Rendkívül gazdag perifériákban a Luminary cég mikrovezérlő-családja is. Egy példát kiragadtam, és ez látható a következő képen:

Figure 1-1. Stellaris® 8000 Series High-Level Block Diagram



Ennek a csipnek az az érdekessége, hogy tartalmaz egy 10/100 Mbit/s sebességű Ethernet interfészt: mind MAC²⁴-et, mind PHY²⁵-t, „kívülről” csak az illesztőtranszformátort kell az áramkörre csatlakoztatni.

2.3. Szoftverfejlesztés ARM processzorra

Az előző részekben az ARM processzorok történetével és általános felépítésével foglalkoztam. Ebben az alfejezetben azt fogom leírni, hogy a kiválasztott ARM Cortex-M3 magot tartalmazó mikrovezérlőkre hogyan lehet szoftvert fejleszteni. Ugyan volt már szó a gyártóspecifikus perifériakészletről, mégis arra koncentrálok, hogy csak a Cortex-M3 lehetőségeit használjam.

2.3.1. C/C++ fordító fordítása ARM architektúrára

A mai mikrovezérlőket már nem érdemes assembly nyelven programozni, mert annyi utasítást és címezési módot kellene ismerni, hogy a befektetett energia nem lenne arányos a várható eredménnyel. Ha mégis szükség lenne egy-egy speciális assembly utasításra, azt beszúrom a C forráskódba.

Másik nyomós érv a – most már nyilvánvalónak tűnő – C/C++ nyelv mellett, hogy a csipgyártó programozói is C nyelven teszik közzé a nem-ARM perifériákat kezelő függvénykönyvtárat (továbbiakban: firmware library).

24 Medium Access Control – közvetítőhozzáférés-vezérlési réteg, I. még: OSI modell 2. rétege

25 Physical Layer – fizikai réteg, I. még: OSI modell 1. rétege

Megfontolás tárgyává téve a dolgot úgy döntöttem, hogy a GNU GCC²⁶ fordítót fogom használni. Az ARM Cortex-M3 csak a Thumb2 utasításkészletet képes végrehajtani, ezért olyan verziójú fordítót kell beszerezni, ami támogatja azt. A GCC esetében ez a 4.3.0, ami a <http://gcc.gnu.org/> címről ingyenesen letölthető. A GCC a binutils-t²⁷ is használja, ezért ezt is le kell tölteni a <http://www.gnu.org/software/binutils/> címről. A harmadik dolog, ami sokat segíthet a GDB, a GNU debugger (letölthető innen: <http://sourceware.org/gdb/>). Ez a program lehetővé teszi, hogy úgy kövessük nyomon a mikrovezérlő működését, hogy közben látjuk a forráskódot.

A C fordító fordítása így történik:

- Először konfiguráljuk a binutils-t úgy, hogy képes legyen ARM-ra fordítani.
- Majd lefordítjuk azt, és installáljuk is.
- Aztán konfiguráljuk a GCC-t is, úgy, hogy ez is ARM-ra fordítson.
- Lefordítjuk ezt is. Ez persze nem lesz zökkenőmentes, mert nem létezik olyan verziójú GCC, ami minden gond nélkül lefordítható.
- Végül konfiguráljuk a GDB-t is, csakúgy, mint a GCC-t.
- Ennek sem a fordítása, sem az installálása nem szokott gondot okozni.

Most pedig álljanak itt a konkrét parancssorok (Linux alatt működnek):

```
tar xzvf binutils-2.18.tar.gz
cd binutils-2.18/
./configure --target=arm-none-linux-gnueabi
make
sudo make install
```

A GCC fordítása kissé bonyolultabb. Érdemes megemlíteni, hogy a libstdc++-v3 sem fordítható le, mert egy-két programozási hiba van benne. Ugyanez igaz majdnem minden függvénykönyvtárra, amit a GCC forrásával adnak. Ezen részek fordítását mindenképpen érdemes letiltani.

Egy másik érdekes kérdés lehet a libc (C függvénykönyvtár) léte vagy nem-léte. Azt gondolom, hogy elég kevés olyan szabványos C függvény van, ami tényleg hasznos egy mikrovezérlő programozásához, ezért libc-t sem fordítok a GCC-vel.

Nyelvek tekintetében kissé érdekes a helyzet. A C nyelv mindenképpen szükséges. De a C++-ról már lehet vitatkozni: vajon van-e olyan alkalmazás, amihez célszerű az objektumorientált szemlélet. Nekem az a véleményem, hogy a C++ számos olyan szolgáltatást tartalmaz, ami megkönnyíti a munkámat: függvények alapértelmezett paramétere, operátor és függvénytúlterhelés, kivételkezelés (try-catch), stb. Ezért a C++ támogatást is fordítok a GCC-hez.

```
tar xjvf gcc-core-4.3.1.tar.bz2
tar xjvf gcc-g++-4.3.1.tar.bz2
cd gcc-4.3.1/
./configure --target=arm-none-linux-gnueabi --enable-languages=c,c++ \
--disable-libstdcxx --disable-libgomp --disable-libmudflap \
--disable-libssp
make
```

²⁶ GNU Compiler Collection - GNU fordítógyűjtemény

²⁷ Tartalmazza az assemblert, linkert, formátum konvertert, stb.

```
sudo make install
```

Ha a `newlib` nevű C függvénykönyvtárt is szeretnénk a GCC-vel együtt lefordítani, akkor másoljuk a `newlib` forrását a `gcc-4.3.1` könyvtárba, és konfigurálásnál a következő parancsot használjuk:

```
./configure --target=arm-none-linux-gnueabi --enable-languages=c,c++ \
--disable-libstdc++ --disable-libgomp --disable-libmudflap \
--disable-libssp --with-newlib
```

A fordítás során fellépő hibák javítását itt nem tudom megadni, a korrigálás meghaladja ezen mű kereteit (másrészt valószínűleg úgyszólván verziófüggők a hibák). A legfontosabb hibaforrások:

- `libgcc`: `BITS_PER_UNIT = 8`
- `libgcc`: header fájlok nem találhatók
- `libgcc`: lebegőpontos számításokhoz szükséges függvények nem kerülnek lefordításra. Ezen úgy lehet segíteni, ha a `libgcc2.h`-ban megadjuk, hogy kell lebegőpontos támogatás a `libgcc`-be.
- `crt`: nem keletkezik CRT²⁸ (C Runtime), ami egyébként nem baj, csak megszakad a fordítás

Ha mindezen túljutottunk, már csak a GDB-t kell lefordítani. Ez már kifejezetten egyszerű folyamat a GCC fordításához képest:

```
tar xjvf gdb-6.8.tar.bz2
cd gdb-6.8
./configure --target=arm-none-linux-gnueabi
make
sudo make install
```

Ezzel el is készült a C/C++ fordító. Ez már majdnem elég ahhoz, hogy programot tudjunk írni ARM Cortex-M3-ra.

2.3.2. C/C++ program írása ARM architektúrára

Az előző részben az olvasható, hogy hogyan kell/kellene a GCC-t és a GDB-t lefordítani, hogy aztán segítségükkel ARM processzort programozhassunk.

A régi időkben a programok forráskódját (legyen az assembly, C vagy C++ nyelvű) egy szövegszerkesztő (text editor) segítségével állították elő (innen ered az elnevezés: az assemblyből fordított végrehajtható (azaz bináris) kódot *szövegnek* (.text) nevezték, hiszen az assembly és a gépi kód majdnem kölcsönösen egyértelműen megfeleltethető). Ezzel a jól bevált hagyománnyal én sem fogok szakítani.

Ahhoz, hogy a munkát el tudjuk kezdeni, néhány dolgot tisztáznunk kell:

²⁸ Olyan függvények, melyek előkészítik a `main()` függvény számára a környezetet (pl. 0-val töltik fel a globális változók memóriaterületét), illetve a `main()` után futnak le (pl. nyitott fájlok lezárása).

Az ARM régebben csak a processzor *szilícium rajzolatát* (layout) és a *felhasználás jogát* (licenz) adta el, de nem foglalkozik a kiegészítő perifériákkal, memóriákkal. Nem adott útmutatást arra vonatkozóan sem, hogy a kiegészítő eszközöket hogyan célszerű a processzorhoz illeszteni.

Az ARM Cortex típusú processzorok esetében a cég szakított ezzel a hagyománnyal, és pontosan specifikált jó néhány paramétert: meghatározta a memóriatérképet, a memóriák (FLASH és statikus RAM) helyét, a kötelező perifériák (megszakításvezérlő, systick²⁹ számláló) regisztereinek címét, az MPU és rendszerregiszterek felépítését, stb.

Adott típusú memóriákban nem csak egyféle információt lehet tárolni: például a FLASH memóriába tölthetünk futtatható kódot vagy konstans adatot, de a RAM-ban lehet inicializált adat (globális vagy statikusnak deklarált lokális változók) vagy veremterület (visszatérési cím, függvényparaméterek, lokális változók) is.

Ezeket a memóriaterületeket szekcióknak nevezzük. A linker, ami összefűzi a firmware (beágyazott rendszer szoftver) komponenseit, nagyban épít a szekció információkra. Tipikusan a következő szekciókat használjuk:

- `.text`: a bináris program (a szöveg)
- `.rodata`: csak olvasható (konstans) adatok
- `.data`: inicializált adatok (globális vagy statikusra deklarált változók)
- `.bss`: inicializálatlan adatok (függvények visszatérési címe,

A felsoroltakon kívül még számos szekció lehetséges (globális objektumok konstruktorait, destruktoraikat kezelő szekció, C programot inicializáló, stb.). Sőt, a későbbiekben szükség is lesz saját szekciók definiálására. Ennek már csak azért is célszerű, mert a C kódban nem lehet megadni egy „objektum” (függvény, változó) címét, míg a szekciók által ez könnyen elérhető.

Ez az a pont, ahol fel kell sorolni az ARM cél által meghatározott memóriaterületek funkcióját, és definiálni kell a szükséges szekciókat (el kell készíteni a linker szkriptet, ami alapján a linker elvégzi a C/C++ program „megfelelő helyre igazítását”):

Név	Címtartomány	Mérete	Eszköz típusa
Code (FLASH)	0x00000000-0x1FFFFFFF	500 MB	Normál (memória)
SRAM	0x20000000-0x3FFFFFFF	500 MB	Normál (memória)
Perifériák	0x40000000-0x5FFFFFFF	500 MB	Gyártóspecifikus
Külső RAM	0x60000000-0x9FFFFFFF	1 GB	Normál (memória)
Külső eszközök	0xA0000000-0xDFFFFFFF	1 GB	Külső eszközök
Perifériák	0xE0000000-0xFFFFFFFF	512 MB	Rendszerezszközök

Ezek közül a legfontosabbak a memóriaterületek, mert a perifériák esetében úgymint pointerok segítségével végezzük az elérést. A memóriák közül is inkább a FLASH érdemel több figyelmet, ugyanis az itt található az az adatok, melyek segítségével inicializálja magát a mikrovezérlő (ez nem gyártóspecifikus, az ARM deklarálta).

A memória legelején egy pointer (32 bites memóriacím található), ami a verem elejére mutat. Fontos tudni, hogy a mai ARM-ok hátulról növekedő vermet használnak, és a veremmutató az utolsó, de valós adatra mutat (nem pedig az első, üres elemre).

A FLASH további 32 bites értékei a megszakítások belépési pontjaira (függvénypointerok – szintén memóriacímek, csak másfélék) mutatnak. Ezt a táblázatot a megszakításvezérlő használja, de a program futása során át lehet helyezni a RAM-ba, de én még sosem tettem ilyet.

Eddig elhallgattam, de tovább nem tehetem: létezik inicializált adatterület, de honnan lesz az inicializálva? A megoldás kézenfekvő: a FLASH-ból, hiszen az „nem felejt el” a tartalmát.

²⁹ Olyan számláló / időzítő, amely segít az operációs rendszerek kialakításában, időalapot szolgáltat az ütemező számára.

Egy lehetséges linker szkript, ami lefedi a veremcímet, a megszakítási vektortáblát, a programkódot, az inicializált adatokat, az inicializálatlan adatokat és a vermet, így néz ki:

```
/* -----
 * Linker script file for ARM Cortex-M3 microcontrollers
 * ----- */

MEMORY
{
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x20000
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

/* Section Definitions */
SECTIONS
{
    /* Code and constant data */
    .text :
    {
        _pointers = .;

        /* Initial value of the stack pointer after RESET */
        *(.stack_pointer)

        /* Pointers to Interrupt (including RESET) service routines */
        *(.vectors)

        /* Code sections */
        _text = .;
        *(.text .text.*)
        /* Read-only data */
        *(.rodata .rodata*)
        _etext = .;
    } > FLASH

    /* Initialized data (read-write) */
    .data : AT (_etext)
    {
        _data = .;
        *(.data .data.*)
        _edata = .;
    } > SRAM

    /* Uninitialized data (heap memory + stack) */
    .bss (NOLOAD) :
    {
        _bss = .;
        *(.bss .bss.*)
        _ebss = .;
    } > SRAM

    . = ALIGN(4);
    _end = .;

    /* Stack will be at the end of the RAM area */
}
```

Ezek után következzen egy fordítható-futtatható programkód bemutatása. Az előbb definiált szekciókat „fel kell tölteni” értelmes adatokkal és kódokkal, ebben pedig a C fordító lesz segítségünkre.

```

/* -----
 * This file contains the startup code for the ARM Cortex microcontroller.
 * ----- */

#include <config.h>
#include <sysinit.h>

/* -----
 * The first word of the FLASH should be the initial stack pointer of the
 * microcontroller.
 * This parameter will be in the ".stack_pointer" section.
 * See also: linker script
 * ----- */

__attribute__((section(".stack_pointer")))
void *stack_pointer = (void *) (MAIN_STACK);

/* -----
 * The next words should be pointers to ISRs (Interrupt Service Routines).
 * These parameters will be placed into the ".vectors" section.
 * See also: linker script
 * ----- */

__attribute__((section(".vectors")))
void (*vectors[])() = { sysinit, 0, 0, 0, 0, 0,
                       0, 0, 0, 0, 0, 0,
                       0, 0, 0, 0, 0
};

/* -----
 * The function will be started after RESET.
 * ----- */

void sysinit() {
    unsigned char *ptr;

    /* Initialize ".data" section with binary 0s */
    for (ptr = (unsigned char *)RAM_BASE; ptr < (unsigned char *) (MAIN_STACK); ptr++)
        *ptr = 0;

    /* Main loop increments a counter */

    for (;;)
        asm("nop");
}

```

A `config.h` fájl néhány előre definiált konstanst tartalmaz, pl. a RAM kezdetét (0x20000000) és a verem címét (0x20000000 + 8 kb-át).

A `stack_pointer` (veremmutató) egy memóriacím, ami a `.stack_pointer` szekcióba kerül (vagyis a memória legelejére, pontosan úgy, ahogy az ARM specifikációjában le van írva).

Ezt követi a `vectors` tömb, ami a megszakítási vektorok címét tartalmazza. Helye a `.vectors` szekcióban van, ami sorrendben a veremmutatót követi (ez a linker szkriptből is kiderül: közvetlenül a veremmutató után szerepel).

Ezután következik az összes programkód. Ezek automatikusan a `.text` szekcióba kerülnek, míg az adatok a `.data` szekcióba.

RESET hatására elindul a 0. megszakítási vektor, vagyis a `sysinit` függvény. Nem csinál ez mást, mint inicializálja (jelen esetben nullákkal tölti fel) a RAM-ot, majd belefut egy végtelen ciklusba. Hogy ez valóban így van-e, le kell fordítani a forráskódot. Mivel ezt többször is el fogom végezni, ezért `Makefile`³⁰-t használok:

```
# -----
# This is the Makefile for ST's STM32 (ARM-Cortex based) Microcontollers
#
# Change CROSS parameter if you want to use a different C/C++ compiler or
# the path to the C/C++ compiler is different.
# -----

CROSS = arm-none-linux-gnueabi-
CC     = $(CROSS)gcc
CXX    = $(CROSS)g++
AS     = $(CC)
OPT    = 1
CFLAGS = -mthumb -mcpu=cortex-m3 -Wall -O$(OPT) -g -I. -I.. -D__STM32__
CXXFLAGS = $(CFLAGS)
LD     = $(CROSS)ld
LDFLAGS = -T cortex_m3.ld

OBJDUMP = $(CROSS)objdump
ODFLAGS = -h -j .stack_pointer -j .vectors -j .text -j .data -j .bss -dS
OBJCOPY = $(CROSS)objcopy
OCFLAGS = -O binary -j .stack_pointer -j .vectors -j .text -j .data -j .bss
NM      = $(CROSS)nm

PROG    = firmware_cortex_m3

# -----
# Core modules of the firmware application
# -----

OBJS    = sysinit.o

# -----
# Compile the firmware
# -----

all: clean $(OBJS)
      $(LD) $(LDFLAGS) -o $(PROG) $(OBJS) $(EXT_LIBS)
      $(OBJDUMP) $(ODFLAGS) $(PROG) > $(PROG).list
      $(OBJCOPY) $(OCFLAGS) $(PROG) $(PROG).bin
      $(NM) $(PROG) | sort > $(PROG).nm

# -----
# Clean unnecessary files
# -----

clean:
      rm -rf $(OBJS) $(PROG) $(PROG).list $(PROG).hex $(PROG).nm $(PROG).bin
```

A fordítás ezek után könnyen elvégezhető: a `make` program teljes mértékben automatizálja a folyamatot. A fordítás eredményeként számos fájl létrejön. Ezek közül a legfontosabb az ELF³¹ formátumú futtatható fájl és a program listája (ami tartalmazza a C forráskódot, a címeket és a generált assembly utasításokat is). A lista fájl tartalma a következő:

30 Fordítást automatizáló program (make) „konfigurációs” fájlja

31 Executable and Linkable Format - futtatható és linkelhető formátum

```

firmware_cortex_m3:      file format elf32-littlearm

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .text          00000060  00000000  00000000  00008000  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .debug_abbrev   0000008d  00000000  00000000  00008060  2**0
                   CONTENTS, READONLY, DEBUGGING
  2 .debug_info     0000009e  00000000  00000000  000080ed  2**0
                   CONTENTS, READONLY, DEBUGGING
  3 .debug_line     0000003e  00000000  00000000  0000818b  2**0
                   CONTENTS, READONLY, DEBUGGING
  4 .debug_frame    00000020  00000000  00000000  000081cc  2**2
                   CONTENTS, READONLY, DEBUGGING
  5 .debug_pubnames 0000003c  00000000  00000000  000081ec  2**0
                   CONTENTS, READONLY, DEBUGGING
  6 .debug_aranges  00000020  00000000  00000000  00008228  2**0
                   CONTENTS, READONLY, DEBUGGING
  7 .debug_str       0000007f  00000000  00000000  00008248  2**0
                   CONTENTS, READONLY, DEBUGGING
  8 .comment         0000002b  00000000  00000000  000082c7  2**0
                   CONTENTS, READONLY
  9 .ARM.attributes 00000031  00000000  00000000  000082f2  2**0
                   CONTENTS, READONLY

Disassembly of section .text:

00000000 <_pointers>:
    0: 20002000      .word  0x20002000

00000004 <vectors>:
    4: 00000041  00000000  00000000  00000000      A.....
    ...

00000040 <sysinit>:

/* -----
 * The function will be started after RESET.
 * ----- */

void sysinit() {
    40: f04f 5200      mov.w   r2, #536870912 ; 0x20000000
        unsigned char *ptr;

        /* Initialize ".data" section with binary 0s */
        for (ptr = (unsigned char *)RAM_BASE; ptr < (unsigned char *) (MAIN_STACK); ptr++)
    44: f242 0300      movw    r3, #8192      ; 0x2000
    48: 4619           mov     r1, r3
    4a: f2c2 0100      movt    r1, #8192      ; 0x2000
        *ptr = 0;
    4e: f04f 0300      mov.w   r3, #0 ; 0x0
    52: f802 3b01      strb.w  r3, [r2], #1
    56: 428a           cmp     r2, r1
    58: d1f9           bne.n  4e <sysinit+0xe>

        /* Main loop increments a counter */

        for (;;)
            asm("nop");
    5a: bf00           nop
    5c: e7fd           b.n    5a <sysinit+0x1a>
    5e: 46c0           nop                                (mov r8, r8)

```

Látható, hogy 0-s címen a RAM végének címe van, ez a verem teteje, a következő (4-es) címen pedig a `sysinit` első utasításának címe található (azért 0x41, mert a legelső bit jelzi, hogy Thumb2 „üzemmódba” kell váltani – az ARM Cortex-M3 csak azt ismeri.)

Ezek után már csak azzal kell megismerkedni, hogy miként lehet az elkészült kódot a mikrovezérlő memóriájába tölteni, futtatni és hibamentesíteni. Ez lesz a következő rész témája.

2.3.3. Kód letöltése, hibamentesítés

ARM alapú mikrovezérlők programozására az OpenOCD-t használható. Ez a nyílt forráskódú program szabadon letölthető a <http://openocd.berlios.de/web/> oldalról. Mivel ez nem annyira egyértelmű, ezért inkább a következő módszert javaslom (előtte a `subversion` programot/csomagot telepíteni kell):

```
svn checkout svn://svn.berlios.de/openocd/trunk
```

A letöltött forráskód birtokában kezdődhet a fordítás. Ha a számítógép, amelyen a programozás történik, rendelkezik beépített (nem USB-s) párhuzamos porttal, ajánlatos a fordítást `--enable-parport` opcióval végezni.

```
cd trunk/
./bootstrap
./configure --enable-parport
make
sudo make install
```

Miközben az előző részeket írtam, vásároltam egy Olimex ARM-USB-OCD típusú JTAG programozót. Nem azért tettem azt, mert az párhuzamos (printer) portos programozó nem használható, hanem azért, mert a párhuzamos port folyamatosan „kopik ki” a számítógépekből.

Ha a programozó eszköz FT232L-vel van felépítve (pl. Olimex ARM-USB-OCD), akkor a `libusb` és a `libftdi` függvénykönyvtárak (vagy inkább az FTDI saját meghajtójának) telepítése után így végezzük a fordítást:

```
cd trunk/
./bootstrap
./configure --enable-ft232l_ftd2xx
make
sudo make install
```

Az OpenOCD futtatásához szükséges egy konfigurációs fájlt létrehozni `openocd.cfg` néven. A név nem kötelező, de ez az alapértelmezett, az OpenOCD ezen a néven keresi.

A konfigurációs fájl a következőket írja le:

- Az OpenOCD a 4444-es telnet porton (TCP) érhető el.
- Ha GDB-t használunk, a 3333-es portra kell csatlakozni azzal.

- Ha **párhuzamos portos** programozót használunk, ami a 0x378-as portcímen található, az `interface parport` és a `parport_port 0x378` használata szükséges.
- Az **FT2232 alapú** eszközök alkalmazása esetén az `interface ft2232` beállítást használjuk. Ebben az esetben meg kell adni a programozó típusát: `ft2232_layout "olimex-jtag"` és az eszköz azonosítóit: `ft2232_vid_pid 0x15ba 0x0003`
- A kábel típusa a párhuzamos portos esetben Wiggler.
(http://wiki.openwrt.org/OpenWrtDocs/Customizing/Hardware/JTAG_Cable)
- A JTAG beállításai (ez egy viszonylag lassú kapcsolatot biztosít, de ha a mikrovezérlő órajel-generátora (kvarc) már elindult, a JTAG sebessége növelhető **és növelendő** 500-1000 kHz-re).
- A csip egy ARM Cortex-M3 típusú, little-endian eszköz.
- A FLASH memória típusa, kezdőcíme, mérete.
- A `working area` (munkaterület) a programozás során pufferként szolgál.

```
#daemon configuration
telnet_port 4444
gdb_port 3333
tcl_port 6666

#interface
interface parport
parport_port 0x378
parport_cable wiggler

jtag_khz 8
jtag_nsrst_delay 10
jtag_ntrst_delay 10

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe
jtag_device 5 0x1 0x1 0x1e

#target configuration
#target <type> <startup mode>
target cortex_m3 little 0

#flash configuration
working_area 0 0x20000000 0x4000 nobackup
flash bank stm32x 0x08000000 0x00008000 0 0 0
```

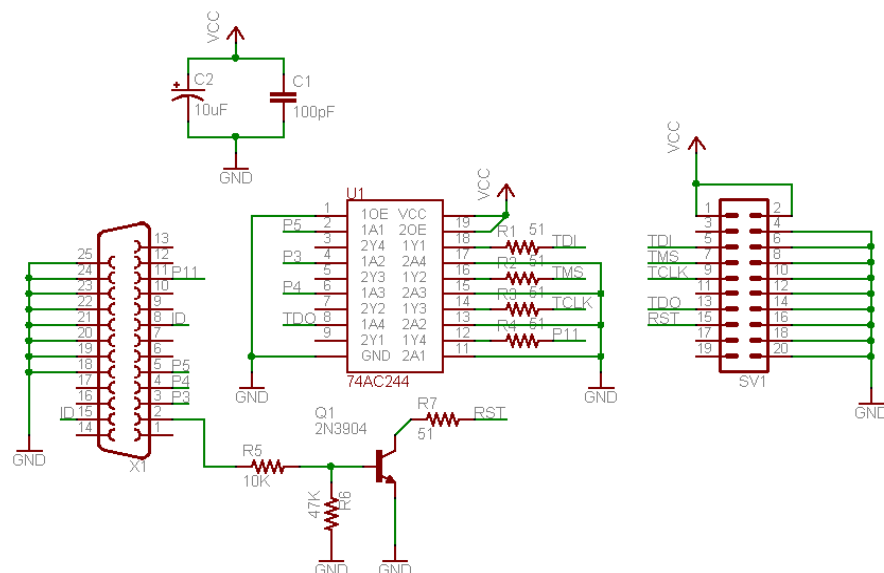
Az FT2232-re épülő programozó konfigurációs fájlja (részlet):

```
...
tcl_port 6666

interface ft2232
ft2232_device_desc "Olimex OpenOCD JTAG A"
ft2232_layout "olimex-jtag"
ft2232_vid_pid 0x15ba 0x0003
```

```
jtag_khz 8
...
```

A programozás megkezdése előtt mindenképpen el kell készíteni vagy meg kell vásárolni a JTAG programozó hardvert. Egy rendkívül egyszerű áramkör felépítését mutatja a következő kapcsolási rajz:



Tovább szoktam egyszerűsíteni az áramkört azzal, hogy kihagyom belőle a 74HC244-et, a tranzisztort és az ellenállásokat. Gyakorlatilag egy kábel marad, amit egy-egy csatlakozó zár le mindkét végén. **Érdemes a TRST-t is bekötni a printer port 6-os lábára.**

A pontos bekötés érdekében érdemes áttanulmányozni a `trunk/src/jtag/parport.c` fájlt, annak is a következő két sorát. Mivel az én számítógépem `nBUSY` lába valamiért nem működik, ezért **az `nBUSY` (7. bit, 11-es láb) bemenetet a `SELECT-re` (4. bit, 13-as láb) cseréltem**, pontosabban a kettőt összekötöttem.

```

/* name      tdo   trst  tms   tck   tdi   srst   o_inv i_inv init  exit  led */
{ "wiggler", 0x80, 0x10, 0x02, 0x04, 0x08, 0x01, 0x01, 0x80, 0x80, 0x80, 0x00 },
/*          nBUSY D4    D1    D2    D3    D0    o_inv → 0x00, ha nincs tranzisztor */

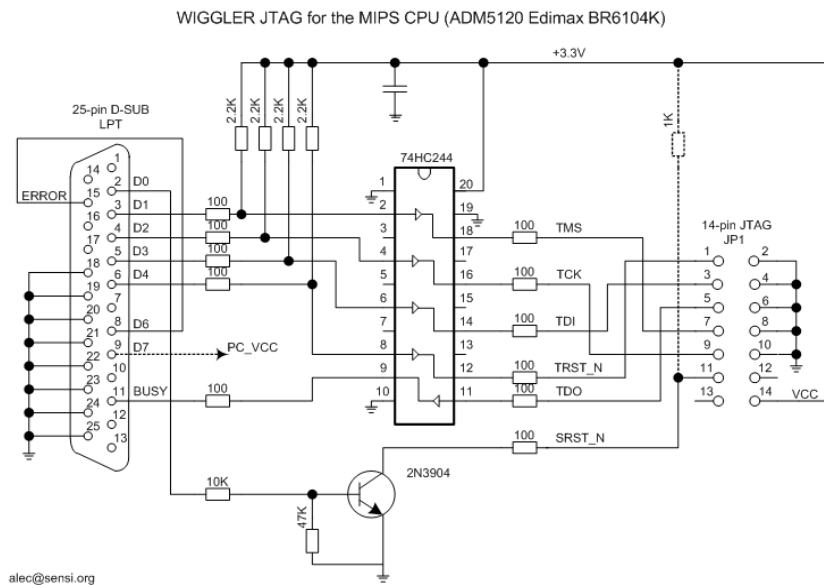
En ezt így módosítottam:

/* name      tdo   trst  tms   tck   tdi   srst   o_inv i_inv init  exit  led */
{ "wiggler", 0x10, 0x10, 0x02, 0x04, 0x08, 0x01, 0x00, 0x00, 0x91, 0x91, 0x00 },
/*          SELECT D4    D1    D2    D3    D0    o_inv → 0x00, ha nincs tranzisztor */

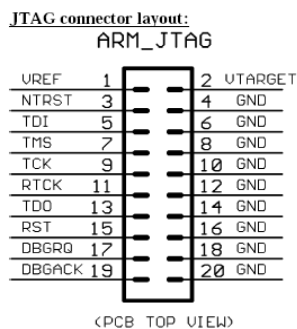
```

A sorok értelmezésére már valószínűleg magától is ráírt az Olvasó.

A következő ábra az OpenWRT oldalán található JTAG kábel bekötését mutatja. Mivel én nem invertálom a `RESET` jelet (tranzisztorral), ezért az előbb leírtak most is érvényesek.



FT2232-vel felépített eszközökön természetesen nem szükséges módosításokat végezni. Akár párhuzamos portos, akár FT2232-es programozót használunk, valamiképpen csatlakozni kell a mikrovezérlőhöz. Az ARM ezt is szabványosította: megadott egy 20 lábú kiosztást, ahogyan a JTAG programozó csatlakozik a processzorhoz.



Ezek után nincs más hátra, mint az előző ábra alapján bekötni a JTAG programozót a mikrovezérlő megfelelő lábaira. Az OpenOCD így indítható:

```
sudo ./openocd
```

Optimális esetben (néhány óra hibakeresés után) így válaszol az OpenOCD:

```
Info: options.c:50 configuration_output_handler(): Open On-Chip Debugger 1.0
(2008-06-27-14:04) svn:734
Info: options.c:50 configuration_output_handler(): jtag_speed: 1, 1
Info: jtag.c:1389 jtag_examine_chain(): JTAG device found: 0x3ba00477 (Manufacturer:
0x23b, Part: 0xba00, Version: 0x3)
Info: jtag.c:1389 jtag_examine_chain(): JTAG device found: 0x16410041 (Manufacturer:
0x020, Part: 0x6410, Version: 0x1)
```

Az OpenOCD használatához be kell „telnetelni” a démonba:

```
telnet localhost 4444
```

Ennek hatására megjelenik a prompt:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
>
```

A flash törlése és írása a következő paranccsal végezhető el:

```
flash write_image erase firmware_cortex_m3 0x8000000
```

A helyes válasz:

```
> flash write_image erase firmware_cortex_m3 0x8000000
auto erase enabled
device id = 0x20006410
flash size = 128kbytes
wrote 96 byte from file firmware_cortex_m3 in 0.593120s (0.158062 kb/s)
>
```

A kód ezek után így futtatható: `reset halt majd resume`:

```
> reset halt
JTAG device found: 0x3ba00477 (Manufacturer: 0x23b, Part: 0xba00, Version: 0x3)
JTAG device found: 0x16410041 (Manufacturer: 0x020, Part: 0x6410, Version: 0x1)
target state: halted
target halted due to debug request, current mode: Thread
xPSR: 0x01000000 pc: 0x00000040
> resume
> poll
target state: running
>
```

A `poll` paranccsal azt ellenőriztem, hogy fut-e a kód. Láthatólag fut.

Ha azt szeretnénk tudni, hogy mi történik futás közben, akkor le kell állítani a kód futását (vagy el sem kell indítani – a RESET ettől függően még így is ajánlatos) a `halt` paranccsal, majd a `step` paranccsal lehet lépegetni utasításról utasításra. A `reg` parancs a mikrovezérlő regisztereit listázza ki.

```
> halt
target state: halted
target halted due to debug request, current mode: Thread
xPSR: 0x61000000 pc: 0x0000005c
> step
target state: halted
target halted due to single step, current mode: Thread
xPSR: 0x61000000 pc: 0x0000005a
> step
target state: halted
target halted due to single step, current mode: Thread
xPSR: 0x61000000 pc: 0x0000005c
> step
target state: halted
target halted due to single step, current mode: Thread
xPSR: 0x61000000 pc: 0x0000005a
> step
target state: halted
target halted due to single step, current mode: Thread
xPSR: 0x61000000 pc: 0x0000005c
> reg
(0) r0 (/32): 0x20000090 (dirty: 0, valid: 1)
(1) r1 (/32): 0x20002000 (dirty: 0, valid: 1)
(2) r2 (/32): 0x20002000 (dirty: 0, valid: 1)
(3) r3 (/32): 0x00000000 (dirty: 0, valid: 1)
(4) r4 (/32): 0x40022010 (dirty: 0, valid: 1)
(5) r5 (/32): 0x4002200c (dirty: 0, valid: 1)
(6) r6 (/32): 0x25d1534c (dirty: 0, valid: 1)
(7) r7 (/32): 0xb52d6fd4 (dirty: 0, valid: 1)
(8) r8 (/32): 0x9feffffdc (dirty: 0, valid: 1)
(9) r9 (/32): 0xdfddb5fe (dirty: 0, valid: 1)
(10) r10 (/32): 0xa4c3ea69 (dirty: 0, valid: 1)
(11) r11 (/32): 0xc9366b88 (dirty: 0, valid: 1)
(12) r12 (/32): 0xffff7ffff (dirty: 0, valid: 1)
(13) sp (/32): 0x20002000 (dirty: 0, valid: 1)
(14) lr (/32): 0xffffffff (dirty: 0, valid: 1)
(15) pc (/32): 0x0000005c (dirty: 0, valid: 1)
(16) xPSR (/32): 0x61000000 (dirty: 0, valid: 1)
(17) msp (/32): 0x20002000 (dirty: 0, valid: 1)
(18) psp (/32): 0x0e42cb58 (dirty: 0, valid: 1)
(19) primask (/32): 0x00000000 (dirty: 0, valid: 1)
(20) basepri (/32): 0x00000000 (dirty: 0, valid: 1)
(21) faultmask (/32): 0x00000000 (dirty: 0, valid: 1)
(22) control (/32): 0x00000000 (dirty: 0, valid: 1)
>
```

Látható, hogy a mikrovezérlő a 0x5a és a 0x5c című utasításokat hajtja végre. Nem meglepő ez: a végtelen ciklust futtat.

```
/* Main loop increments a counter */

for (;;)
    asm("nop");
5a: bf00          nop
5c: e7fd          b.n    5a <sysinit+0x1a>
```

Az OpenOCD egyik leghasznosabb tulajdonsága az, hogy töréspontokat helyezhetünk el a kódban. A törésponthoz érve a mikrovezérlő felfüggeszti a kód futtatását, és átadja a vezérlést a `Debug` (nyomkövető)

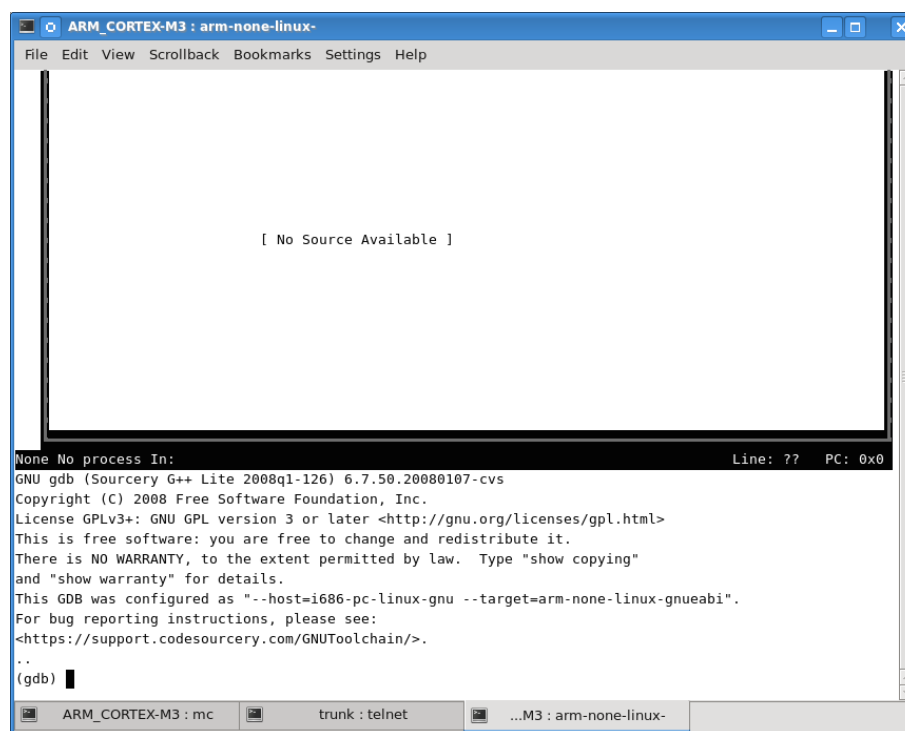
alrendszernek. Ebben az állapotban – a már megismert módon – ellenőrizhetjük a regiszterek értékét, a memóriatartalmakat, és folytathatjuk a kód futtatását. Töréspont elhelyezésére a `bp` parancs szolgál. Paraméterként meg kell adni azt a címet, ahol a kód futását meg kívánjuk állítani, és be kell állítani a töréspont hosszát (ez utóbbi paraméterről nincsenek részletes információim).

```
> bp 0xf0 0
breakpoint added at address 0x000000f0
> resume
target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x000000f0
>
```

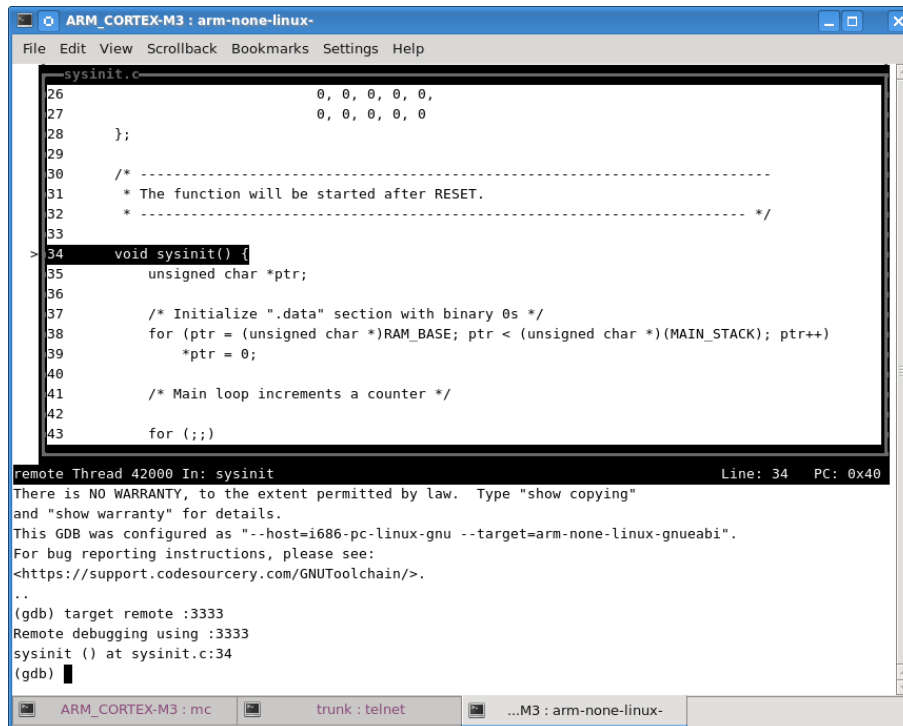
Ha magasabb (azaz C nyelvű) nyomkövetésre van szükség, ennek sincs akadálya: csak el kell indítani az `arm-none-linux-gnueabi-gdbtui-t`, paraméterként átadva a firmware fájl nevét:

```
arm-none-linux-gnueabi-gdbtui firmware_cortex_m3
```

Ezután ajánlatos RESET-elni a mikrovezérlőt, hogy az alapállapotba kerüljön (különösen a DEBUG áramkörre).



A „`target remote :3333`” paranccsal csatlakoztatom a GDB-t az OpenOCD-hez. Ekkor betöltődik a forráskód a GDB felső ablakába.



The screenshot shows the ARM_Cortex-M3 IDE interface. The top window displays the `sysinit.c` file with the following code:

```
26         0, 0, 0, 0, 0,
27         0, 0, 0, 0, 0,
28     };
29
30     /* -----
31     * The function will be started after RESET.
32     * ----- */
33
34 void sysinit() {
35     unsigned char *ptr;
36
37     /* Initialize ".data" section with binary 0s */
38     for (ptr = (unsigned char *)RAM_BASE; ptr < (unsigned char *) (MAIN_STACK); ptr++)
39         *ptr = 0;
40
41     /* Main loop increments a counter */
42
43     for (;;)
44         ;
45 }
```

The bottom window shows the GDB console output:

```
remote Thread 42000 In: sysinit Line: 34 PC: 0x40
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi".
For bug reporting instructions, please see:
<https://support.codesourcery.com/GNUToolchain/>.
..
(gdb) target remote :3333
Remote debugging using :3333
sysinit () at sysinit.c:34
(gdb) 
```

Töréspontot a `b` (breakpoint – töréspont) paranccsal lehet definiálni. Paraméterként elfogadja a memóriacímet és a függvénynevet is. A futtatást a `c` (continue – folytatás) paranccsal lehet kezdeni / folytatni. A kód futása a töréspontnál megáll. Lépésenként való futtatásra az `n` (next – következő) és az `s` (step – léptetés) parancs használható. Az előbbi a függvényhívásokat egyben végrehajtja, míg az utóbbi elugrik a hívott függvény törzséhez, és lépésenként hajtja végre azt.

Egy változó értékének megtekintése a `p` (print – nyomtatás) paranccsal lehetséges.

```
target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x01000000 pc: 0x0000004e
(gdb) p ptr
$1 = (unsigned char *) 0x20000000 ""
(gdb)
```

Hasznos lehet a programok futását assembly nyelven nyomon követni. Ezt úgy érhetjük el, hogy a `layout asm` parancsot adjuk a GDB-nek.

```

0xe4 <sysinit>      push    {r4, r5, lr}
0xe6 <sysinit+2>     sub     sp, #12
0xe8 <sysinit+4>     bl      0x9d8 <clock_enable_main_osc+20>
0xec <sysinit+8>     bl      0x9c4 <clock_enable_main_osc>
> 0xf0 <sysinit+12>   bl      0xa68 <gpio_init+20>
0xf4 <sysinit+16>    mov.w   r0, #115200 ; 0xc200
0xf8 <sysinit+20>    bl      0xd08 <usart_init+20>
0xfc <sysinit+24>    movs    r0, #89
0xfe <sysinit+26>    bl      0xbdc <CAN_init+20>
0x102 <sysinit+30>   movs    r0, #0
0x104 <sysinit+32>   mov     r1, r0
0x106 <sysinit+34>   mov     r2, r0
0x108 <sysinit+36>   bl      0xb58 <CAN_set_filter+20>
0x10c <sysinit+40>   ldr     r2, [pc, #100] (0x174 <sysinit+144>)
0x10e <sysinit+42>   ldr     r3, [pc, #104] (0x178 <sysinit+148>)
0x110 <sysinit+44>   movs    r1, #0
0x112 <sysinit+46>   str     r1, [r3, #0]
0x114 <sysinit+48>   str     r1, [r2, #0]
0x116 <sysinit+50>   ldr     r0, [pc, #100] (0x17c <sysinit+152>)

remote Thread 42000 In: sysinit                                     Line: 83
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi".
For bug reporting instructions, please see:
<https://support.codesourcery.com/GNUToolchain/>.
..
(gdb) layout asm
(gdb) n
The program is not being run.
(gdb) target remote :3333
Remote debugging using :3333
sysinit () at sysinit.c:83
(gdb)

```

Termékekben lehetőségünk van visszatérni a C nyelvű nyomkövetéshez: használjuk a `layout prev` parancsot.

```

--sysinit.c
73 void sysinit() {
74     // Structure to configure Interrupt Controller
75     NVIC_InitTypeDef NVIC_InitStructure;
76
77     // Enable main (Quartz) oscillator
78     clock_enable_main_osc();
79     // Enable PLL: see also PLL_MUL and PLL_DIV constants
80     clock_enable_pll();
81
82     // Low level initialization of the GPIO ports
> 83     gpio_init();
84     // Low level initialization of the USART
85     usart_init(USART_BAUDRATE);
86     // Low level initialization of the CAN
87     CAN_init(f_cpu / 32 / CAN_BAUDRATE - 1);
88     CAN_set_filter(0, 0, 0);
89
90     // Initialize global variables (flags and queues)
91     cr_needed = line_len = 0;

remote Thread 42000 In: sysinit                                     Line: 83
For bug reporting instructions, please see:
<https://support.codesourcery.com/GNUToolchain/>.
..
(gdb) layout asm
(gdb) n
The program is not being run.
(gdb) target remote :3333
Remote debugging using :3333
sysinit () at sysinit.c:83
(gdb) layout prev
(gdb)

```

Ha azt gondolja az Olvasó, hogy elég nehézkes két program egyidejű használata ugyanarra a célra, jól gondolja. Szerencsére a GDB programozói gondoltak erre az esetre is: a `mon` (monitor) parancs a paraméterként kapott sztringet átadja az OpenOCD-nek az pedig úgy hajtja végre, mintha közvetlenül annak adtuk volna ki a parancsot: a következő kép azt mutatja, hogy mi történik a `mon mdw 0 64` utasítás hatására – azaz írja ki a memória tartalmát a 0. címtől 64 bájt hosszan.

```

Serial-CAN : arm-none-linux-
File Edit View Scrollback Bookmarks Settings Help

sysinit.c
73 void sysinit() {
74     // Structure to configure Interrupt Controller
75     NVIC_InitTypeDef NVIC_InitStructure;
76
77     // Enable main (Quartz) oscillator
78     clock_enable_main_osc();
79     // Enable PLL: see also PLL_MUL and PLL_DIV constants
80     clock_enable_pll();
81
82     // Low level initialization of the GPIO ports
83     gpio_init();
84     // Low level initialization of the USART
85     usart_init(USART_BAUDRATE);
86     // Low level initialization of the CAN
87     CAN_init(f_cpu / 32 / CAN_BAUDRATE - 1);
88     CAN_set_filter(0, 0, 0);
89
90     // Initialize global variables (flags and queues)
91     cr_needed = line_len = 0;

remote Thread 42000 In: sysinit Line: 83
(22) control (/32): 0x00000000 (dirty: 0, valid: 1)
(gdb) mon mdw 0 64
0x00000000: 20002000 000000e5 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1
0x00000020: 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 0000076d
0x00000040: 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1
0x00000060: 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1
0x00000080: 000000e1 000000e1 000000e1 000000e1 00000561 000000e1 000000e1 000000e1
0x000000a0: 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1
0x000000c0: 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1 000000e1
0x000000e0: 46c0e7fe b083b530 fc76f000 fc6af000 fcbaf000 30e1f44f fe06f000 f0002059
(gdb)
Serial-CAN : mc ...AN : arm-none-linux- /t : bash

```

Ezzel elérkeztünk a fordítás, a linkelés, a letöltés és a nyomkövetés tárgyalásának végére. Az eddig elmondottak minden ARM Cortex-M3 mikrovezérlőre igazak (kivéve az OpenOCD memória konfigurációja, az sajnos gyártófüggő).

Az OpenOCD és a GDB parancsait nem tudom olyan részletességgel ismertetni, mint szeretném, de elég sok dokumentáció található az interneten mindkét programról.

2.4. Gyártóspecifikus hardver kezelése

Az előbbi fejezetben arra koncentráltam, hogy az előbbi szoftver csak az ARM Cortex-M3 képességeit használja ki. Ebben a részben azt mutatnám meg, hogy hogyan lehet a gyártó saját függvénykönyvtárát használni szoftverfejlesztésre. Ez a függvénykönyvtár (idegen nyelven: firmware library) segít kezelni azon perifériákat, melyeket a gyártó az ARM mag mellett kialakított. Mivel a perifériák kezelése nem mondható egyszerűnek, ezért a gyártó sokszor előre megírt függvényeket biztosít a felhasználó számára. A diplomamunka további részében is ezt a függvénykönyvtárat fogom használni.

Az elkészítendő áramkör egy LED-et fog villogtatni a PORT B 15-ös lábán. A villogás időalapját a már említett SYSTICK timer lesz, ami megszakítást generál adott időközönként.

Mivel a programfejlesztés módját (forráskód írása, linker szkript, fordítás, OpenOCD kezelése, hibamentesítés) az előző fejezetben már ismertettem, így ettől most eltekintek. A forráskódot érintő változás leginkább a `sysinit.c`-ben jelentkezik:

Megjelenik a SYSTICK időzítő megszakítási rutinjának címe a megszakítási vektortáblában:

```

__attribute__((section(".vectors")))
void (*vectors[])() = {

```

```

sysinit, no_handler, no_handler, no_handler, no_handler,
no_handler, no_handler, no_handler, no_handler, no_handler,
no_handler, no_handler, no_handler, no_handler, systick,
no_handler, no_handler, no_handler, no_handler, no_handler,

```

Ezen kívül számos inicializáló függvényt hív meg a `sysinit()` függvény:

```

// -----
// The function will be started after RESET.
// -----

void sysinit() {
    // Enable main (Quartz) oscillator
    clock_enable_main_osc();
    // Enable PLL: see also PLL_MUL and PLL_DIV constants
    clock_enable_pll();

    // Low level initialization of the GPIO ports
    gpio_init();

    // Initialization of the SysTick Timer
    // Parameter: period time: 1/n sec, where "n" is the parameter
    systick_init(4);

    // Finally, the main function will be started
    while (1)
        main();
}

```

Először engedélyezi a főoszillátort (`clock_enable_main_osc()`), majd beállítja a PLL³² szorzó és osztó értékeit, és engedélyezi a PLL-t (`clock_enable_pll()`). Mindkét függvény a firmware library függvényeit használja (helyük: `stm32/clock.c`):

```

// -----
// This function enables the "main" (Quartz) oscillator
// -----

int clock_enable_main_osc() {
    ErrorStatus HSEStartUpStatus;

    /* RCC system reset (for debug purpose) */
    RCC_DeInit();

    /* Enable HSE */
    RCC_HSEConfig(RCC_HSE_ON);

    /* Wait till HSE is ready */
    HSEStartUpStatus = RCC_WaitForHSEStartUp();

    if (HSEStartUpStatus == SUCCESS) {
        /* Enable Prefetch Buffer */

```

32 Phase Locked Loop – fáziszárt hurok, frekvenciaszorzásra használjuk. A kvarcoszcillátor 12 MHz-es jelét 72 MHz-re konvertálja.

```

    FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

    /* HCLK = SYSCLK */
    RCC_HCLKConfig(RCC_SYSCLK_Div1);

    /* PCLK2 = HCLK */
    RCC_PCLK2Config(RCC_HCLK_Div1);

    /* PCLK1 = HCLK */
    RCC_PCLK1Config(RCC_HCLK_Div1);

    /* Select HSE as system clock source */
    RCC_SYSCLKConfig(RCC_SYSCLKSource_HSE);

    /* Wait till HSE is used as system clock source */
    while(RCC_GetSYSCLKSource() != 0x04);

    return 0;
}

return 1;
}

// -----
// This function enables the PLL.
// Input parameters are: PLL divisor, PLL multiplier
// The CPU frequency is: f_quartz * PLL_multiplier / PLL_divisor
// -----

int __clock_enable_pll(unsigned int divisor, unsigned int multiplier) {
    FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);

    /* Flash 2 wait state */
    FLASH_SetLatency(FLASH_Latency_2);

    /* HCLK = SYSCLK */
    RCC_HCLKConfig(RCC_SYSCLK_Div1);

    /* PCLK2 = HCLK */
    RCC_PCLK2Config(RCC_HCLK_Div1);

    /* PCLK1 = HCLK/2 */
    RCC_PCLK1Config(RCC_HCLK_Div2);

    /* PLLCLK = 8MHz * 9 = 72 MHz */
    RCC_PLLConfig(divisor, multiplier);

    /* Enable PLL */
    RCC_PLLCmd(ENABLE);

    /* Wait till PLL is ready */
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);

    /* Select PLL as system clock source */
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);

    /* Wait till PLL is used as system clock source */
    while(RCC_GetSYSCLKSource() != 0x08);

    return 0;
}

// -----

```

```
// This function enables the PLL.
// The CPU frequency is: f_quartz * PLL_multiplier / PLL_divisor
// -----

int clock_enable_pll() {
    __clock_enable_pll(PLL_DIV, PLL_MUL);
    return 0;
}
```

A fenti két függvény a firmware library része. Működésüket jelen műben nem tudom kifejteni, mert a területi határok erősen kötnek. Ennek ellenére nem okoz hátrányt az áttanulmányozásuk, így ugyanis ízelítőt kapunk abból ami ténylegesen lejátszódik egy „gyári” függvény hívásakor. A többi, gyártótól kapott függvény is hasonlóan működik.

Általában elmondható a gyártó függvényeiről, hogy kétféleképpen fogadnak paramétereket:

- vagy **függvény-argumentumként** (ezt láttuk eddig),
- vagy egy **struktúrát kell „kitölteni”**. Ez utóbbi arra ad lehetőséget, hogy
 - bizonyos paraméterek értékét előre inicializálja egy megfelelő eljárás, és nekünk csak a ténylegesen megváltoztatandó értékeket kelljen módosítani,
 - másrészt több (akár 10-20) paramétert adjunk ár úgy, hogy még mindig áttekinthető maradjon a programunk.

Ez utóbbira mutat példát a `gpio_init()` függvény megvalósítása (`stm32/gpio.c`-ben található). A `gpio_init()` függvényt a `sysinit()` hívja meg, hogy beállítsa a PORTB-t úgy, hogy azon keresztül LED-et tudjunk villogtatni:

```
// -----
// This function initializes the PORT B port.
// -----

void gpio_init() {
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
}
```

A SYSTICK időzítő beállítása, ami a `sysinit()` függvényben történik, a az első módszert alkalmazza:

```
// -----
```

```
// This function initializes the SYSTICK timer. The period contains the
// "FREQUENCY" of the timer interrupt
// -----

int systick_init(unsigned int freq) {
    SysTick_SetReload(CLOCK_FREQ * PLL_FACTOR / freq / 8);
    SysTick_ITConfig(ENABLE);
    SysTick_CounterCmd(SysTick_Counter_Enable);

    return 0;
}
```

A továbbiakban sem tudom majd kifejtetni a firmware library függvényeit, de megpróbálom érthető és logikus sorrendben megmutatni a forrásukat.

Ha az előbbi inicializálásokat elvégezzük, a SYSTICK időzítő megszakítási rutinja (`systick()`) periodikusan meghívódik. Ebben a függvényben kell a LED állapotát váltogatni: ha eddig nem világított, akkor be kell kapcsolni, ha eddig világított, akkor ki kell kapcsolni. Ezt teszi az `irq.c`-ben található `systick()` függvény:

```
// -----
// PORT B 15 (blinking)
// -----

#define LED_SYSTICK (1 << 15)

// -----
// Stores the value of the "blinking" LED.
// -----

volatile unsigned int counter;

// -----
// ISR of the SYSTICK timer (makes the LED blinking).
// -----

void systick() {
    counter++;

    if (counter % 2 == 1) {
        gpio_set(LED_SYSTICK);
    } else {
        gpio_clear(LED_SYSTICK);
    }
}
```

A `gpio_set()` és a `gpio_clear()` egy-egy függvény, feladatuk, hogy a megfelelő port lábak értékét „1”-be, illetve „0”-ba állítsák.

Fordításkor és linkeléskor fel kell sorolni a program fájljai mellett a firmware library fájljait is (pl. a `Makefile`-ban):

```
# -----
```



```

# This is the Makefile for ST's STM32 (ARM-Cortex based) Microcontollers
#
# Change CROSS parameter if you want to use a different C/C++ compiler or
# the path to the C/C++ compiler is different.
# -----

CROSS  = arm-none-linux-gnueabi-
CC      = $(CROSS)gcc
CXX     = $(CROSS)g++
AS      = $(CC)
OPT     = 3
FWLIB   = ../stm32/src
DRIVERS = ../stm32
CFLAGS  = -mthumb -mcpu=cortex-m3 -Wall -O$(OPT) -g -DSTM32
CFLAGS += -I$(FWLIB)/../inc -I. -I$(DRIVERS)
CXXFLAGS = $(CFLAGS)
LD       = $(CROSS)ld
LDFLAGS  = -T stm32.ld

OBJDUMP  = $(CROSS)objdump
ODFLAGS  = -h -j .stack_pointer -j .vectors -j .text -j .data -j .bss -dS
OBJCOPY  = $(CROSS)objcopy
OCFLAGS  = -O binary -j .stack_pointer -j .vectors -j .text -j .data -j .bss
NM       = $(CROSS)nm

# -----
# Name of the program (firmware)
# -----

PROG    = blinky

# -----
# Core modules of the firmware application
# -----

OBJS     = sysinit.o main.o irq.o

# -----
# Hardver drivers (Hardver abstraction layer and Software library)
# -----

# -----
# HAL modules:
# -----

OBJS     += $(DRIVERS)/clock.o $(DRIVERS)/systick.o $(DRIVERS)/gpio.o

# -----
# Software library modules:
# -----

OBJS     += $(FWLIB)/stm32f10x rcc.o $(FWLIB)/stm32f10x flash.o
OBJS     += $(FWLIB)/stm32f10x systick.o $(FWLIB)/stm32f10x gpio.o

# -----
# Compile the firmware
# -----

all: clean $(OBJS)
      $(LD) $(LDFLAGS) -o $(PROG) $(OBJS) $(EXT_LIBS)
      $(OBJDUMP) $(ODFLAGS) $(PROG) > $(PROG).list
      $(OBJCOPY) $(OCFLAGS) $(PROG) $(PROG).bin
      $(NM) $(PROG) | sort > $(PROG).nm

```

```
# -----  
# Clean unnecessary files  
# -----  
  
clean:  
    rm -rf $(OBJ) $(PROG) $(PROG).list $(PROG).hex $(PROG).nm $(PROG).bin
```

Az előbb bemutatott (és valójában nagyon egyszerű) program arra szolgált, hogy segítse a `firmware library` függvényeinek megértését. Nem mellékes az sem, hogy sikerült az egyik legegyszerűbb és legelegánsabb hardver komponens, a SYSTICK időzítőt beállítani és használni. A következő fejezetben egy összetettebb áramkör, egy teljes soros-CAN átalakító tervezésével, építésével és programozásával ismerkedhet meg az Olvasó. Hozzá kell tennem, hogy a soros-CAN átalakító építéséhez számos háttérinformációra lesz szükség, (amennyiben ezen mű terjedelme ezt lehetővé teszi) ezeket közölni fogom a jobb megértés és kevesebb utánaolvasás érdekében.

3. Soros-CAN átalakító

Ebben a fejezetben azt ismerhetjük meg, hogy milyen elve, módszerek felhasználásával lehet a legegyszerűbben soros-CAN átalakítót készíteni. A teljes forrás közzétételére nincsen lehetőség, terjedelmi okok korlátoznak ebben.

A soros-CAN átalakító feladata, hogy az USART³³-on érkező parancsok segítségével módosíthassuk az átalakító működését, a CAN busz sebességét, a szűrők beállításait, és üzeneteket küldhessünk-fogadhassunk a CAN buszon keresztül. A soros porton zajló kommunikáció teljesen szöveges alapú, így egy tetszőleges soros terminál vagy terminál emulátor program segítségével használható. A felhasználó felület programja is karakteres üzemmódban kommunikál az átalakítóval.

A soros port beállításai: **1200 8N1**, kézfogásos üzemmód (handshaking) nélkül. Később ez változhat, de a mostani verzió még nem támogatja ezt.

3.1. Hardver kialakítása

Az átalakító „lelke” egy ARM Cortex-M3 alapú mikrovezérlő, amely több soros porttal (USART) és egy CAN vezérlővel rendelkezik. Az egyszerűség kedvéért a legelső soros porton (USART1) történik a kapcsolattartás a vezérlő számítógéppel, ami tipikusan a felhasználó személyi számítógépe, vagy egy olyan beágyazott eszköz, amely nem rendelkezik (a ma már jogosan elvárható) CAN vezérlővel.

Az átalakító másik kommunikációs interfésze a CAN buszra csatlakozik. A CAN vezérlő beállítása a soros porton keresztül lehetséges.

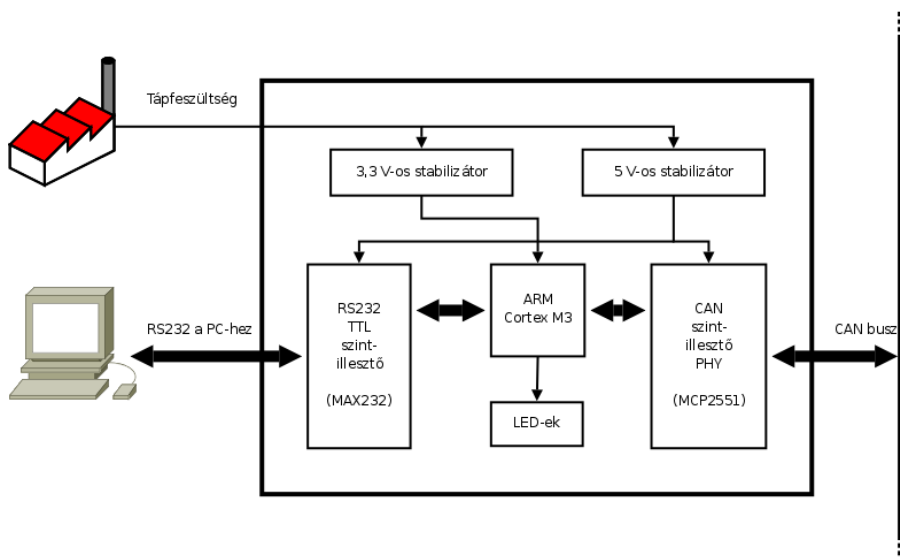
Mielőtt megismerkednénk a konkrét megvalósítással, nézzük meg az átalakító általános felépítését és a kétféle busz működését. Terjedelmi korlátok miatt nem lehetséges, hogy a buszokról túl sok információt közöljek, így mindenképpen biztatnám az Olvasót, hogy keressen és olvasson további szakmai leírásokat interneten.

3.1.1. Blokkvázlat, kapcsolási rajz

Ebben a részben azt nézzük meg, hogy miképpen célszerű kialakítani a soros-CAN átalakító hardver összetevőit.

A következő blokkvázlat az átlagosnál részletesebb lesz, ennek az az oka, hogy könnyebb legyen a később bemutatandó kapcsolási rajz értelmezése.

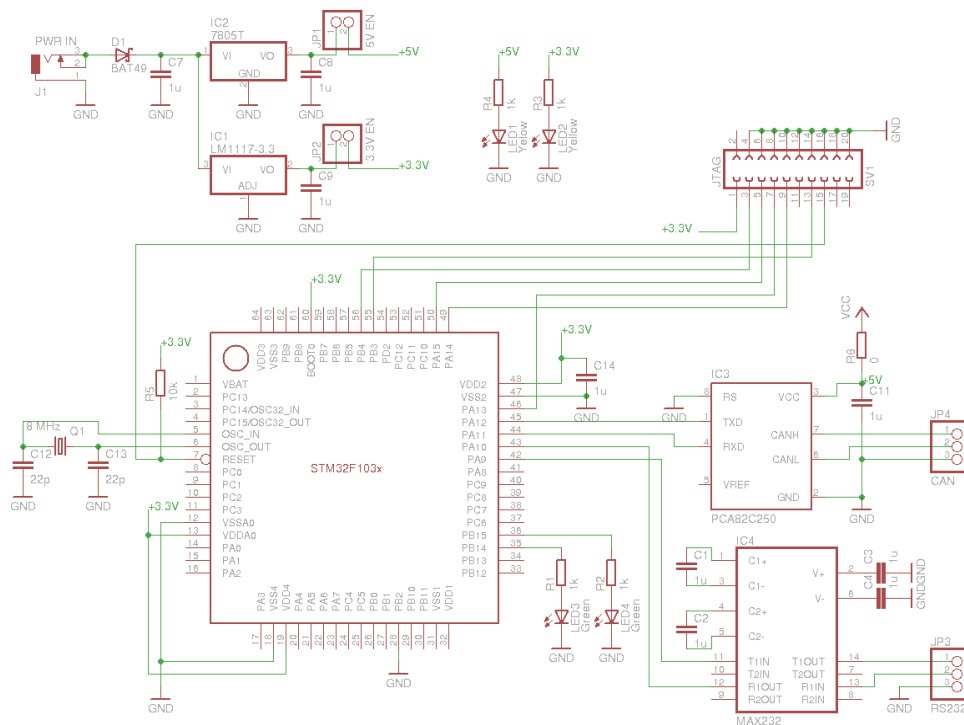
33 Universal Synchronous / Asynchronous Receiver Transmitter – univerzális szinkron-aszinkron adó-vevő



Az RS232-TTL szintillesztő arra szolgál, hogy a +15...-15 V-os jeleket TTL (5 V) szintű feszültséggé alakítsa. A mikrovezérlő kimenetei és bemenetei 3,3 V-ról járnak, ami azért nem okoz problémát, mert a jelszinteket úgy definiálták, hogy legyen bőven átfedés a kétféle rendszer között.

A CAN illesztő is 5 V-ról működik, így ezt az 5 V-os stabilizátorra kell kötni. Természetesen létezik 3,3 V-os változata a CAN és az RS232 illesztőnek, de ebbe az áramkörbe az 5 V-os példányok kerültek.

A LED-et közvetlenül a mikrovezérlő táplálja. Ezt azért tehetjük meg, mert a LED-ek már 5-10 mA-es áramnál is megfelelő fényességgel világítanak. A LED-ek lehetőséget adnak arra, hogy a mikrovezérlő állapotáról tájékozódjunk.

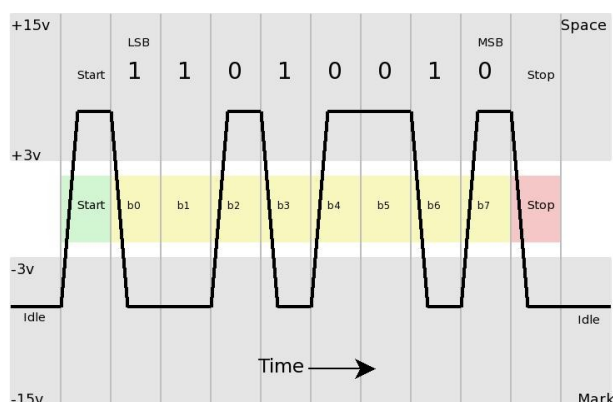


3.1.2. Az RS232 rövid áttekintése

Az RS232-t a 60-as években találták ki arra, hogy a soros terminálokat a számítógéphez kapcsolják. Alapvetően pont-pont kapcsolatot biztosít, karakteres átvitelt tesz lehetővé opcionális paritásbit ellenőrzéssel. A kommunikáció sebessége beállítható, csakúgy, mint az átvitt adatok hossza. A beállításokat egy elegáns sztringgel szoktuk megadni: 115200 8N1. Ez azt jelenti, hogy 11500 bit/sec sebességet használunk, 8 bites adathosszal, paritásellenőrzés nélkül (no parity), 1 stop bittel.

Mivel pont-pont kapcsolatról van szó, ezért a kommunikációhoz 3 vezetékre van szükség: egy adatvonal a számítógép és az átalakító között („odafelé”), egy adatvonal az átalakító és a számítógép között („visszafelé”), és egy földvezeték. Ez utóbbi azért szükséges, hogy legyen egy közös referenciapotenenciál.

Az RS232 által használt fizikai jelfolyamot (idődiagramot) és az alkalmazott feszültségszinteket a következő ábra mutatja:



A busz akkor szabad, ha az adott adatvezeték *idle* (logikai 1) állapotban van. A karakter küldését stop bittel (logikai 0 szint) kezdi a küldő fél. Ezután következnek az adatbitek, először az LSB kerül átvitelre. Az adatbitek számát a paraméterek beállításakor adjuk meg (5-9 bit). Ebben az esetben 7 adatbitet küldünk. Ha van paritásbit, az az adatbitek követi. Lehetőség van páros, páratlan, jel (*mark*) és üres (*space*) bit küldésére is. A karakter átvitelének végét a stop bit jelzi, ez kötelezően logikai 1 értékű. Ha ez nem logikai 1, akkor keretezési hiba (*framing error*) keletkezik. Ezt használja ki a LIN busz a keret (adatkapcsolati réteg kerete) kezdetének jelzésére. A stop bitek száma 1, 1,5 vagy 2 lehet.

Mivel az RS232 oda-vissza jelvezetékekkel rendelkezik, csak pont-pont kapcsolat kialakítására alkalmas. Az idők során számos zseniális megoldás született arra, hogy több eszközt köthessünk össze busz topológiát használva. A leginkább kedvelt busz az RS485, amit mind a mai napig előszeretettel használnak az iparban. A keretezés (értsd: adatkapcsolati réteg) módját nem írja le a szabvány, így meglehetősen rugalmas hálózatot alakíthatunk ki. Hátránya, hogy nem egyszerű jó adatkapcsolati réteget kialakítani RS485-höz. Keretezésre legtöbbször HDLC-t, MODBUS-t vagy saját megoldást használnak. Magam is kitaláltam egy kicsit erőforrásigényes, de nagyon flexibilis módszert, mely lehetőséget biztosít arra, hogy egymással teljesen inkompatibilis eszközök osztozzanak ugyanazon a vezetékpáron.

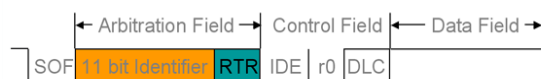
3.1.3. A CAN busz működése

A CAN buszt a Bosch cég kezdte fejleszteni 1983-ban az autókban alkalmazott kábelkorbácsok leváltására. Azóta az ipar számos területén használják, mert nagyon megbízható, és jelentős mértékben tehermentesíti a mikrovezérlőt. Legfontosabb tulajdonságai a következők:

- A szabvány definiálja fizikai jelszinteket, a huzalozás módját (**szimmetrikus** érpár, maximum 1 Mbit/sec sebességgel, **open kollektoros** meghajtással).
- Szabványos keretformátumot (összeköttetésmentest) használ.
- A magasabb prioritású üzenetek előnyt élveznek a kevésbé fontosabbakkal szemben (versengés [arbitráció] során dől el).
- Lehetőség van a bejövő üzenetek (hardveres) szűrésére, ezzel tehermentesül a mikrovezérlő.
- Maximum 8 adatbájtot visz át egy lépésben.
- 16 bites CRC-t használ az adat integritásának ellenőrzésére.
- Megbízható összeköttetést biztosít.
- Az újabb eszközök támogatják a régebbi eszközöket.
- Nem használ címeket a forrás- és a céleszköz azonosítására (de nem is tiltja meg), **hanem az üzeneteket azonosítja.**

A CAN buszon közlekedő keretek formátumát a következő ábra mutatja:

Standard Frame Format

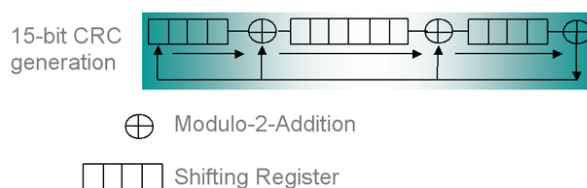


Extended Frame Format

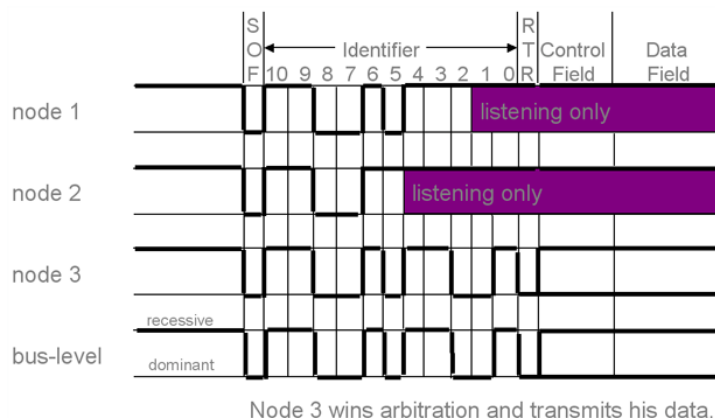


Láthatjuk, hogy a busz akkor tekinthető szabadnak, ha adott ideig egyik eszköz sem hajtja meg logikai 0-val.

A keret kezdetét egy SOF (start of frame) bitkombináció jelzi. Ezután következik annak eldöntése, hogy melyik adó ragadhatja magához a busz használatát. Ennek eldöntésére versengést (arbitráció) használnak. A magasabb prioritású üzenetek azonosítója (identifier) (ami 11 vagy 29 bit hosszúságú lehet) „hamarabb” tartalmaz logikai 0-s bitet, mint az alacsonyabbaké. Az alacsonyabb prioritású üzenet küldője elveszti a versengést, ha logikai 1-et küld akkor, amikor a magasabb prioritású üzenet küldője 0-t. A vesztes vezérlő figyelő (vevő) üzemmódba vált. Erre láthatunk példát a következő ábrán:



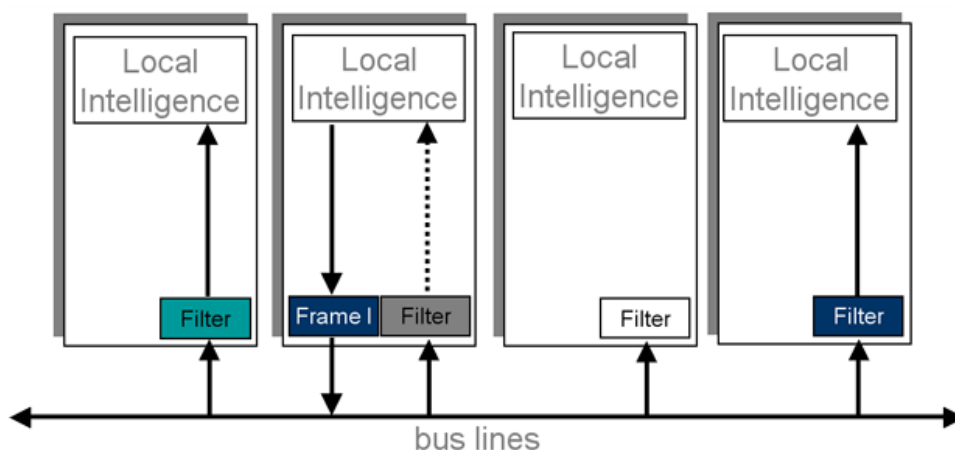
$$X^{15} + X^{14} + X^{10} + X^8 + X^7 + X^4 + X^3 + 1$$



Az ábrán azt látjuk, hogy a 2-es állomás elveszti a versengést, amikor a 4. bitpozíción logikai 1-est küld, míg a többi vezérlő 0-t. Az 1. állomás az 1. bitpozíción veszíti el az arbitrációt, mert a 3. állomás 0-t küld. Ezután a 3. állomás jogosult a buszt használni, a többi vezérlő figyelő állapotba (listening only) kerül.

Az arbitráció részt néhány vezérlő bit, majd az adatbájtok számának átvitele követi. Az adatbájtok (max. 8 db) küldése után egy 16 bites CRC-t is ad a nyertes fél, hogy a vevő leellenőrizhesse az üzenet épségét.

Sikeres vétel esetén a keret végén egy nyugta bit érkezik attól a vevőtől, amelyik magáénak érezte a csomagot. A keret lezárása egy megfelelő bitkombinációval történik. A következő ábrán azt látjuk, hogy négy darab CAN buszra kötött eszköz közül a második (balról) éppen ad, míg a többi vesz.



Sikertelen vétel esetén az adó újra megpróbálkozik az üzenet elküldésével, elvileg ezzel jelentősen csökkentheti a busz hatékonyságát. Az általam megírt modul lehetővé teszi, hogy az adó csak véges sokszor (alapesetben ez 16) próbálkozzon, majd jelezzon hibát a „magasabb szoftver rétegek” számára. Ez a megoldás nagyon szépen látszik oszcilloszkópon.

A CAN szabványa nem csak a keretek formátumát határozza meg, hanem a keretek típusait is. Előírás szerint 4-féle keret létezik:

- **Adatkeret** (data frame) amely minimum 0, maximum 8 adatbájtot tartalmazhat (DLC tartalmazza az aktuális értéket)
- **Távoli keret** (remote frame), melyet arra használunk, hogy a vevőtől adatokat kérjünk. Az RTR bit azt mutatja, hogy távoli keretről van szó.
- **Hibakeret** (error frame): hiba esetén keletkezik.

- **Túlerheltség keret** (overload frame), ez akkor érkezik, ha a vevő bemeneti tárolója (RX FIFO) megtelt.

A CAN üzenetek vétele hasonlóan érdekes, mint az adás. Az arbitráció győztese küldi az üzenetet, de melyik állomás veszi? Nos, a válasz egyszerű: minden vevő állomás rendelkezik legalább egy maszkkal (*mask*) és legalább egy azonosítóval (*identifier*). A vevő a buszon megjelenő ÜZENET azonosítója és a maszk között bitenkénti **ÉS** kapcsolatot képez, és ha az eredmény megegyezik a saját azonosítójával, akkor az üzenetet nyugtázza, és a bemeneti tárolójába (RX FIFO) másolja. Ha egy üzenet érkezik, a CAN vezérlő megszakítást generál(hat).

Az előbb elmondottak úgy foglalhatók össze, hogy ahol a maszk értéke '1', ott a CAN vezérlő „figyeli” az ID-k értékét, és ha minden „figyelt” helyen megegyezik a két azonosító, akkor veszi a keretet. Ahol a maszk (és persze a beállított szűrő azonosító) értéke '0', ott az üzenet azonosítója don't care.

A fenti ábra éppen azt az állapotot mutatja, hogy az első és a negyedik állomás veszi azt az üzenetet, amit a második ad.

3.2. Szoftver megoldások

Ebben a részben megismerkedhetünk azokkal az elvekkel és megoldásokkal, melyek szükségesek, de legalábbis hasznosak egy többszálú program fejlesztéséhez. Azért nevezem a soros-CAN átalakító szoftvert többszálúnak, mert bizonyos feladatok egymással látszólag párhuzamosan futnak, így pontosan ugyanazon problémák merülnek fel, mint egy többfeladatos operációs rendszer és a hozzá tartozó programok írásakor.

Az egyik legfontosabb dolog, ami a segítségünkre lehet, az üzenetsor, idegen nyelven ezt *queue*-nak mondják. Ezek olyan adatstruktúrák, melyek lehetővé teszik, hogy egymással párhuzamosan futó folyamatok üzeneteket küldjenek egymásnak úgy, hogy az üzenetek sorrendje nem változik.

Az ezt követő alfejezetben azt nézzük meg, hogy hogyan tudjuk az STM32 soros és CAN interfészét használni. Ez előbb említett üzenetsorok segítségével üzenetet válthatnak a megszakítási rutinok és a főprogram.

A soros-CAN szoftver megoldásának alapgondolata ugyanis az, hogy minden, kívülről érkező esemény megszakítást váltson ki, és a megszakítási rutin erre valamiképpen reagálni tud. A külső események a következők: karakter érkezése a soros portról és keret érkezése a CAN interfészen. A megfelelő megszakítási rutin feladata, hogy az érkezett adatokat feldolgozza, és üzenetsoron keresztül értesítse a főprogramot. A főprogram pedig eldönti, hogy az adott eseményre hogyan kell reagálni: paraméter állítás vagy üzenetküldés szükséges-e.

Fontos alaptétel, hogy ebben a megoldásban csak a főprogram jogosult adatot küldeni (soros porton és CAN-en), míg az adatok vétele a megszakítási rutinokra van bízva. Mivel a főprogram üzeneteket kap az USART és a CAN megszakítástól, így a megszakítási rutinoknak bizonyos mértékig fel kell dolgozniuk a beérkezett adatokat, hogy üzenetté tudják konvertálni azokat.

3.2.1. Üzenetsorok (queue-k)

Az üzenetsorok lényege, hogy készítünk egy adatszerkezetet, ami leginkább a futószalaghoz hasonlít. A küldő fél üzeneteket küldhet az üzenetsor elejére, a vevő pedig üzeneteket olvas ki az üzenetsorból. A futószalagos példával élve ez olyan, hogy az adó adatokat helyez a futószalag egyik végére, a vevő pedig a másik végén hozzáférhet azokhoz.

Persze egy üres üzenetsorból nincs mit kiolvasni és egy tele levő üzenetsorba nem lehet újabb elemet beszúrni. Ezt is figyelembe kell venni a programozás során.

Lássuk először, hogy ebben az esetben milyen az üzenetek formátuma (*queue.h*):


```
// -----
// Message contains a command and three parameter value
// -----

typedef struct {
    unsigned int command;
    unsigned int param1;
    unsigned int param2;
    unsigned int param3;
} t_message;
```

Amint ez látható, az üzenetünk 4 darab egész számot ($4 \times 32 \text{ bit} = 16 \text{ bájt}$) tartalmaz, ezek közül az elsőt a parancs kódjának tárolására használjuk, a másik 3 változót pedig paraméternek használjuk. Pontos jelentésüket a környezet, az aktuális szituáció határozza meg.

A teljes üzenetsor (potenciális) üzenetek sorozata, melyet így deklarálunk:

```
// -----
// If the queue size is not defined, the default value is 1 message/queue
// -----

#ifndef QUEUE_SIZE
#define QUEUE_SIZE    1
#endif

// -----
// The queue contains at least 1 message and the two pointers
// -----

typedef struct {
    t_message data[QUEUE_SIZE];
    unsigned int rd_ptr, wr_ptr;
} t_queue;
```

Az üzenetsor tartalmaz legalább egy (ez a `QUEUE_SIZE` értékétől függ, ami alapesetben 1) üzenetet, és két indexet, melyek azt mutatják, hogy hova lehet a következő üzenetet beszúrni (`wr_ptr`: write pointer, író mutató), illetve honnan lehet a következő üzenetet kiolvasni (`rd_ptr`: read pointer, olvasó mutató).

Ezen az adatszerkezeten a következő műveleteket végezhetjük el:

- Inicializálhatjuk az üzenetsort (paraméterként meg kell adni a sorra mutató pointert):

```
// -----
// Function to initialize the queue
// The parameter is the pointer to the queue
// -----

void queue_init(t_queue *);
```

- Új adatot küldhetünk a sorba: ha a sor már tele van, akkor hibakóddal tér vissza. Paraméterként a sorra mutató pointert és az üzenet pointerét várja:

```
// -----  
// Copies a message to the queue, if it is not full  
// (return value is QUEUE_OK).  
// If the queue is full, the return value will be QUEUE_FULL.  
// Parameters: pointer to the queue, pointer to the message structure  
// -----  
  
t_queue_result queue_put_non_blocking(t_queue *, t_message *);
```

- A következő üzenetre mutató pointert kérhetünk az üzenetsorból. Ha nincs újabb üzenet, vagyis a sor üres, akkor hibakóddal tér vissza.

```
// -----  
// Gets the pointer to the next message in the queue, if it is not empty  
// (return value is QUEUE_OK).  
// If the queue is empty, the return value will be QUEUE_EMPTY.  
// Parameters: pointer to the queue, POINTER TO A POINTER_to_the_message  
// structure.  
// -----  
  
t_queue_result queue_get_non_blocking(t_queue *, t_message **);
```

- Ez a függvény törli a következő üzenetet az üzenetsorból. Az következő üzenet pointerének elkérése azért nem törli az üzenetet, mert így nem kell az üzenetet kimásolni a sorból, hanem „helyben” feldolgozható az, majd ha már nincsen szükség az üzenetre, a következő függvénnyel törölhető az:

```
// -----  
// Removes the last message from the queue.  
// Parameter: pointer to the queue.  
// If the queue is not empty the return value is QUEUE_OK, else QUEUE_EMPTY.  
// -----  
  
t_queue_result queue_remove_non_blocking(t_queue *);
```

Ez előbbi függvények non-blocking függvények voltak, ez azt jelenti, hogy nem akasztják meg a program futását, ha beszúráskor már tele van az üzenetsor, illetve ha olvasáskor nincs mit olvasni. Természetesen létezik blocking (megakasztó) verziója is az előbbi függvényeknek:

```
t_queue_result queue_put(t_queue *, t_message *);  
t_queue_result queue_get(t_queue *, t_message **);
```

```
t_queue_result queue_remove(t_queue *);
```

Lássuk most, hogy az előbbi függvényeket hogyan tudjuk kialakítani: először vegyük szemügyre az alapbeállításokat elvégző függvényt (`queue.c`):

```
// -----
// Function to initialize the queue
// The parameter is the pointer to the queue
// -----

void queue_init (t_queue *queue) {
    queue->rd_ptr = 0;
    queue->wr_ptr = 0;
}
```

Nem csinál ez mást, csak 0 értékűre állítja az író és az olvasó pointer értékét.

```
// -----
// Copies a message to the queue, if it is not full
// (return value is QUEUE_OK).
// If the queue is full, the return value will be QUEUE_FULL.
// Parameters: pointer to the queue, pointer to the message structure
// -----

t_queue_result queue_put_non_blocking(t_queue *queue, t_message *msg) {
    int k;

    // If the rd_ptr == wr_ptr, the queue is empty.
    // If the rd_ptr != wr_ptr, and both of them point to the same item,
    // the queue is full.
    if ((queue->wr_ptr != queue->rd_ptr) && ((queue->wr_ptr % QUEUE_SIZE)
        == (queue->rd_ptr % QUEUE_SIZE))) {
        return QUEUE_FULL;
    } else {
        // Copy the message into the queue
        for (k = 0; k < sizeof(t_message); k++)
            ((char *)&(queue->data[queue->wr_ptr % QUEUE_SIZE]))[k] =
                ((char *)msg)[k];

        // increment write pointer
        queue->wr_ptr++;

        return QUEUE_OK;
    }
}
```

Az üzenet beszúrását végző függvény már valamivel összetettebb. Első lépésben megnézi, hogy lehet-e újabb üzenetet beszúrni. Ha nem, akkor hibakóddal tér vissza, ha lehet, akkor bemásolja az üzenetet az üzenetsorba, majd növeli az író pointer értékét, és pozitív nyugtával tér vissza. Az feltételvizsgálat azért ilyen komplikált, mert különbséget kell tenni aközött, hogy mikor az író és olvasó pointer (moduló üzenetek száma) megegyezik, a sor üres, vagy tele van-e.

A kiolvasó függvény nagyon hasonlít ehhez:

```
// -----  
// Gets the pointer to the next message in the queue, if it is not empty  
// (return value is QUEUE_OK).  
// If the queue is empty, the return value will be QUEUE_EMPTY.  
// Parameters: pointer to the queue, POINTER TO A POINTER_to_the_message  
// structure.  
// -----  
  
t_queue_result queue_get_non_blocking(t_queue *queue, t_message **msg) {  
    if (queue->rd_ptr == queue->wr_ptr) {  
        return QUEUE_EMPTY;  
    } else {  
        *msg = &(queue->data[queue->rd_ptr % QUEUE_SIZE]);  
        return QUEUE_OK;  
    }  
}
```

A függvény ellenőrzi, hogy az üzenetsorban van-e üzenet. Ha nincsen, akkor hibakóddal tér vissza. Ellenkező esetben visszaadja a következő üzenetre mutató pointert.

```
// -----  
// Removes the last message from the queue.  
// Parameter: pointer to the queue.  
// If the queue is not empty the return value is QUEUE_OK, else QUEUE_EMPTY.  
// -----  
  
t_queue_result queue_remove_non_blocking(t_queue *queue) {  
    if (queue->rd_ptr == queue->wr_ptr) {  
        return QUEUE_EMPTY;  
    } else {  
        queue->rd_ptr++;  
        return QUEUE_OK;  
    }  
}
```

Az üzenet törlését végző függvény sem túl komplikált: ha van olyan üzenet, amit lehet törölni, akkor elvégzi az eltávolítást, és pozitív nyugtával tér vissza, ellenkező esetben egy hibakód lesz a válasz.

A blokkoló változatok annyiban különböznek, hogy addig hajtják végre a nem blokkoló változatokat, amíg hibakód érkezik visszatérési értékként.

```
// -----  
// The same as "queue_put_non_blocking()", but this function waits while  
// the queue is full.  
// -----  
  
t_queue_result queue_put(t_queue *queue, t_message *msg) {  
    while (queue_put_non_blocking(queue, msg) == QUEUE_FULL);  
    return QUEUE_OK;  
}  
  
// -----
```

```
// The same as "queue_get_non_blocking()", but this function waits while
// the queue is empty.
// -----

t_queue_result queue_get(t_queue *queue, t_message **msg) {
    while (queue_get_non_blocking(queue, msg) == QUEUE_EMPTY);
    return QUEUE_OK;
}

// -----
// The same as "queue_remove_non_blocking()", but this function waits while
// the queue is empty.
// -----

t_queue_result queue_remove(t_queue *queue) {
    while (queue_remove_non_blocking(queue) == QUEUE_EMPTY);
    return QUEUE_OK;
}
```

Az üzenetsorok ezen megvalósítása azért nagyon elegáns, mert a beszűrő függvény csak az író mutatót változtatja meg, a törölő függvény pedig csak az olvasási mutatót módosítja. Vagyis nem lehetséges az, hogy ugyanazon változó értékét egyszerre két folyamat (főprogram, megszakítások) módosítsa.

3.2.2. Az STM32 soros (RS-232) interfészének használata

Az STM32 soros portjának (USART) programozását a firmware library használatával oldhatjuk meg legegyszerűbben. Korábban láttuk már, hogy a firmware library függvényeit úgy hívjuk meg, hogy paraméterként egy „kitöltött” struktúrát adunk át nekik, mely tartalmaz minden információt, mely a hardver eszköz használatához szükséges. Nincsen ez másként a soros port esetében sem. A kommunikáció paramétereinek beállítását így végezhetjük el (`stm32/usart.c`-ből):

```
// -----
// Initializes the USART controller using a baud rate value
// -----

int usart_init(unsigned int baudrate) {
    USART_InitTypeDef USART_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable USART1, GPIOA and AFIO clocks */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA |
        RCC_APB2Periph_AFIO, ENABLE);

    /* Configure USART1 Tx (PA.09) as alternate function push-pull */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* Configure USART1 Rx (PA.10) as input floating */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* USART and USART2 configured as follow:
```

```
- BaudRate = 1200 baud
- Word Length = 8 Bits
- One Stop Bit
- Even parity
- Hardware flow control disabled (RTS and CTS signals)
- Receive and transmit enabled
*/
USART_InitStructure.USART_BaudRate = baudrate;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

/* Configure USART1 */
USART_Init(USART1, &USART_InitStructure);

/* Enable the USART1 */
USART_Cmd(USART1, ENABLE);

return 0;
}
```

Általánosságban elmondható, hogy két dologra szüksége van minden perifériának: egyrészt tápfeszültséggel el kell látni, mert a táp alapértelmezésben ki van kapcsolva (energiatakarékossági okokból), és órajelet is kell biztosítanunk minden perifériának. Ha a periféria használja a mikrovezérlő lábait, akkor be kell állítani azt is, hogy melyik láb legyen bemenet illetve kimenet. Ezt követi a kommunikációs paraméterek beállítása. Ezzel az inicializálás végéhez értünk.

Soros porton való adatküldéshez nem kell kitölteni semmilyen struktúrát, hiszen a két paraméter, amit meg kell adnunk a következő:

- mit akarunk küldeni,
- és melyik porton keresztül.

(stm32/usart.c)

```
// -----
// Sends a character through USART.
// -----

int usart_send_char(char ch) {
    while(USART_GetFlagStatus(USART1, USART_FLAG_TC) == RESET);
    USART_SendData(USART1, ch);

    return 0;
}

int usart_send_char_non_blocking(char ch) {
    if (USART_GetFlagStatus(USART1, USART_FLAG_TC) == RESET);
    return 0x8000000;

    USART_SendData(USART1, ch);

    return 0;
}
```

Látható, hogy rendelkezésre áll mind a blocking, mind a non-blocking függvény.

A következő kódrészlet az ST firmware libraryéből való, és azt szemlélteti, hogy milyen egyszerűek és elegánsak a függvénykönyvtár elemei (stm32/src/stm32f10x_usart.c)

```

/*****
* Function Name : USART_SendData
* Description : Transmits single data through the USARTx peripheral.
* Input : - USARTx: Select the USART or the UART peripheral.
*         This parameter can be one of the following values:
*         - USART1, USART2, USART3, UART4 or UART5.
*         - Data: the data to transmit.
* Output : None
* Return : None
*****/
void USART_SendData(USART_TypeDef* USARTx, uint16_t Data)
{
    /* Check the parameters */
    assert_param(IS_USART_ALL_PERIPH(USARTx));
    assert_param(IS_USART_DATA(Data));

    /* Transmit Data */
    USARTx->DR = (Data & (uint16_t)0x01FF);
}

```

Egy sztring küldése úgy történik, hogy a karaktereket egymás után átadjuk a soros portnak, küldés céljából (usart.c):

```

// -----
// Sends a character string through USART.
// -----

int usart_send_str(char *str) {
    unsigned int counter;

    for (counter = 0; str[counter]; counter++)
        usart_send_char(str[counter]);

    return 0;
}

```

Mint arról már volt szó, az karakter fogadása megszakítást vált ki, és a kiszolgáló rutin számos funkciót lát el: átveszi a karaktert, ellenőrzést, konverziót, stb. végez, és szükség esetén üzenetet küld a főprogramnak. A karakter átvétele (kiolvasása a vételi bufferből) a következőképpen történik (usart.c):

```

// -----
// Receives a character through USART.
// -----

char usart_rcv_char() {
    while(USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
    return USART_ReceiveData(USART1);
}

```

A függvény visszatérési értéke képviseli a fogadott karaktert. A firmware library megfelelő függvénye a következő:

```

/*****
* Function Name   : USART_ReceiveData
* Description     : Returns the most recent received data by the USARTx peripheral.
* Input          : - USARTx: Select the USART or the UART peripheral.
*                 This parameter can be one of the following values:
*                 - USART1, USART2, USART3, UART4 or UART5.
* Output         : None
* Return         : The received data.
*****/
u16 USART_ReceiveData(USART_TypeDef* USARTx)
{
    /* Check the parameters */
    assert_param(IS_USART_ALL_PERIPH(USARTx));

    /* Receive Data */
    return (u16)(USARTx->DR & (u16)0x01FF);
}

```

Ebben a fejezetben áttekintettük azon függvényeket, melyekkel beállíthatjuk a soros port kommunikációs paramétereit, küldhetünk és fogadhatunk adatokat. A következő alfejezetben a CAN vezérlő hasonló függvényeivel ismerkedhetünk meg.

3.2.3. A CAN busz interfész használata

Mint ahogyan a 3.1.3. fejezetben láthattuk, az STM32 beépített CAN vezérlővel rendelkezik. Ezt a perifériát nagyrészt ugyanúgy használhatjuk, mint a soros portot. Természetesen a beállított paraméterek teljesen el fognak térni, mind számban, mind típusban, de azt általánosságban elmondhatjuk, hogy a CAN periféria kezeléséhez is szükségesek inicializáló függvények, lehetőséget kell biztosítani keretek küldésére és fogadására.

A CAN kontroller alapbeállításai a következő függvényekkel végezhetőek el:

```

// -----
// Initializes CAN controller using a prescaler value
// -----

int CAN_init(unsigned int prescaler);

// -----
// Doesn't initialize the controller, just changes the prescaler value.
// -----

int CAN_set_prescaler(unsigned int prescaler);

// -----
// Sets the filter mask (Filter: the number if the filter to be used [0..13],
// mask and ID are 29-bit values (0x00000000 ... 0xffffffff),
// the MSB is the RTR bit.
// -----

void CAN_set_filter(unsigned int filter, unsigned int mask, unsigned int id);

```


Az első függvény paraméterként a processzor (pontosabban a busz) órajelgenerátorának előosztási értékét (prescaler) várja. Ez egy olyan szám, amely a processzor órajelfrekvenciájából (f_{cpu}) és CAN busz kívánt sebességéből (can_baudrate) számítható az alábbi módon:

```
prescaler = f_cpu / 32 / can_baudrate - 1
```

A CAN_init függvény a következő műveleteket végzi el:

```
// -----
// CAN tries to send messages. If it was not successful, the CAN MAC tries
// to send it again and again... 16 times.
// -----

#ifndef CAN_RETRIES
#define CAN_RETRIES 16
#endif

// -----
// Initializes CAN controller using a prescaler value
// -----

int CAN_init(unsigned int prescaler) {
    GPIO_InitTypeDef GPIO_InitStructure;
    CAN_InitTypeDef  CAN_InitStructure;

    /* CAN register init */
    CAN_DeInit();
    CAN_StructInit(&CAN_InitStructure);

    /* CAN Periph clock enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    /* Configure CAN pin: RX */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* Configure CAN pin: TX */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* CAN cell init */
    CAN_InitStructure.CAN_TTCM = DISABLE;
    CAN_InitStructure.CAN_ABOM = DISABLE;
    CAN_InitStructure.CAN_AWUM = DISABLE;
#ifdef CAN_NO_AUTORETRANSMIT
    CAN_InitStructure.CAN_NART = ENABLE;
#else
#warning Use gcc ... -DCAN_NO_AUTORETRANSMIT ...
    CAN_InitStructure.CAN_NART = DISABLE;
#endif
    CAN_InitStructure.CAN_RFLM = DISABLE;
    CAN_InitStructure.CAN_TXFP = DISABLE;
#ifdef CAN_LOOPBACK
```

```

    CAN_InitStructure.CAN_Mode = CAN_Mode_LoopBack;
#else
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
#endif
CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;
CAN_InitStructure.CAN_BS1 = CAN_BS1_8tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_7tq;
CAN_InitStructure.CAN_Prescaler = prescaler & 0x3ff;
CAN_Init(&CAN_InitStructure);

    return 0;
}

```

Először engedélyezi az órajelet a CAN vezérlő és az általános célú bemenetek/kimenetek (GPIO) számára. Ezután beállítja, hogy a CAN_TX láb kimenetként funkcionáljon, és a CAN_RX láb pedig bemenetként működjön.

Végül átadja a megfelelő paramétereket a CAN vezérlő alacsony szintű konfigurációs függvényeinek, természetesen a már megismert módon.

A fenti függvény érdekessége, hogy az automatikus újraküldés lehetőségének tiltása (CAN_NO_AUTORETRANSMIT) mellett lehetővé teszi, hogy a nyomkövetési (debug) célból a lapkán belül összekössük az adás és a vételi lábat, vagyis a vevő pontosan azt fogja venni, amit az adó ad. Ezt úgy érhetjük el, hogy a CAN_LOOPBACK makrót definiáljuk valahol (például a Makefile-ban).

A következő, beállítást végző függvény a CAN_set_prescaler. Ez a CAN_init-tel ellentétben nem végzi el a teljes inicializálást, csak az előosztó értékét módosítja, melynek hatására változik a CAN kommunikáció sebessége.

A CAN_set_prescaler függvény forráskódja a következő:

```

// -----
// Doesn't initialize the controller, just changes the prescaler value.
// -----

int CAN_set_prescaler(unsigned int prescaler) {
    CAN_InitTypeDef  CAN_InitStructure;

    /* CAN register init */
    CAN_StructInit(&CAN_InitStructure);

    /* CAN cell init */
    CAN_InitStructure.CAN_TTCM = DISABLE;
    CAN_InitStructure.CAN_ABOM = DISABLE;
    CAN_InitStructure.CAN_AWUM = DISABLE;
#ifdef CAN_NO_AUTORETRANSMIT
    CAN_InitStructure.CAN_NART = ENABLE;
#else
#warning Use CAN_NO_AUTORETRANSMIT
    CAN_InitStructure.CAN_NART = DISABLE;
#endif
    CAN_InitStructure.CAN_RFLM = DISABLE;
    CAN_InitStructure.CAN_TXFP = DISABLE;
#ifdef CAN_LOOPBACK
    CAN_InitStructure.CAN_Mode = CAN_Mode_LoopBack;
#else
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
#endif
    CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;
}

```

```

    CAN_InitStructure.CAN_BS1 = CAN_BS1_8tq;
    CAN_InitStructure.CAN_BS2 = CAN_BS2_7tq;
    CAN_InitStructure.CAN_Prescaler = prescaler & 0x3ff;
    CAN_Init(&CAN_InitStructure);

    return 0;
}

```

Látható, hogy ebben a függvényben elmarad a CAN interfész alacsony szintű inicializálása. Szorosan hozzátartozik a paraméterek beállításához a vételi maszk(ok) és az azonosító(k) megadása. Erre a következő két függvény alkalmas:

```

// -----
// Sets the filter mask (Filter: the number if the filter to be used [0..13],
// mask and ID are 29-bit values (0x00000000 ... 0x1fffffff),
// the MSB is the RTR bit.
// -----

void CAN_set_filter(unsigned int filter, unsigned int mask, unsigned int id);

```

Argumentumként meg kell adni, hogy melyik szűrő (*filter*) maszk illetve azonosító értékét szeretnénk megváltoztatni, majd meg kell adnunk a megfelelő maszkot (*mask*) és azonosítót (*id*). Ha egyszer beállítunk egy értékpárt, bármikor módosíthatjuk azt. Ezzel nő a rendszer rugalmassága. A legfelső bittel (MSB) azt állíthatjuk be, hogy a vevő érzékeny legyen-e távoli (remote) keretek vételére, vagy ne foglalkozzon ezzel.

A `CAN_set_filter` forráskódja a következő:

```

// -----
// Sets the filter mask (Filter: the number if the filter to be used [0..13],
// mask and ID are 29-bit values (0x00000000 ... 0x1fffffff),
// The MSB is the RTR bit.
// -----

void CAN_set_filter(unsigned int filter, unsigned int mask, unsigned int id) {
    CAN_FilterInitTypeDef CAN_FilterInitStructure;

    unsigned int new_id = (id << 3) | (id >> 30) | CAN_ID_EXT;
    unsigned int new_mask = (mask << 3) | (mask >> 30) | CAN_ID_EXT;

    /* CAN filter init */
    CAN_FilterInitStructure.CAN_FilterNumber = filter;
    CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask;
    CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;
    CAN_FilterInitStructure.CAN_FilterIdHigh = new_id >> 16;
    CAN_FilterInitStructure.CAN_FilterIdLow = new_id & 0xffff;
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh = new_mask >> 16;
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = new_mask & 0xffff;
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment = CAN_FIFO0;
    CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
    CAN_FilterInit(&CAN_FilterInitStructure);
}

```

Az első néhány sorban kissé meg kell változtatni a bitek sorrendjét, mert a hardver más sorrendben várja, mint ahogyan azt a függvénynek megadjuk. Ezt követi az alacsony szintű firmware library függvény meghívása.

Keretek küldésére is találunk megfelelő függvényt:

```
// -----  
// Sends a CAN message. The "id" determines the ID of the message (29 bits),  
// "length" contains the number of data bytes to be sent,  
// "data_h" and "data_l" are 32 bit values = 8 bytes of data.  
// -----  
  
int CAN_send(unsigned int id, unsigned int length, unsigned int data_h,  
             unsigned int data_l, unsigned int rtr);
```

Ez a függvény nagyban megkönnyíti a CAN keretek küldését, mert csak megadjuk az „cél” állomás azonosítóját, a küldendő adatbájtok számát, magukat az adatbájtokat két darab 32 bites számra szétszedve, valamint azt, hogy a küldendő üzenet távoli keret-e. A függvény visszatérési értéke `CANTXOK` (gyakorlatilag 0), ha sikeres volt a küldés, illetve egy ettől eltérő érték, ha nem sikerült 16 próbálkozás után sem. A függvény forráskódja a következő:

```
// -----  
// Sends a CAN message. The "id" determines the ID of the message (29 bits),  
// "length" contains the number of data bytes to be sent,  
// "data_h" and "data_l" are 32 bit values = 8 bytes of data.  
// -----  
  
int CAN_send(unsigned int id, unsigned int length, unsigned int data_h,  
             unsigned int data_l, unsigned int rtr) {  
  
    unsigned char TransmitMailbox;  
    unsigned int retval = -1, i = 0;  
  
    /* transmit */  
    TxMessage.ExtId = id;  
  
    if (rtr)  
        TxMessage.RTR = CAN_RTR_REMOTE;  
    else  
        TxMessage.RTR = CAN_RTR_DATA;  
  
    TxMessage.IDE = CAN_ID_EXT;  
    TxMessage.DLC = length;  
    TxMessage.Data[0] = (data_h >> 24) & 0xff;  
    TxMessage.Data[1] = (data_h >> 16) & 0xff;  
    TxMessage.Data[2] = (data_h >> 8) & 0xff;  
    TxMessage.Data[3] = (data_h >> 0) & 0xff;  
    TxMessage.Data[4] = (data_l >> 24) & 0xff;  
    TxMessage.Data[5] = (data_l >> 16) & 0xff;  
    TxMessage.Data[6] = (data_l >> 8) & 0xff;  
    TxMessage.Data[7] = (data_l >> 0) & 0xff;  
  
    // Try to send message  
    do {  
        TransmitMailbox = CAN_Transmit(&TxMessage);  
  
        // Waits until the result arrives.  
        if (TransmitMailbox != CAN_NO_MB)
```

```

        while ((retval = CAN_TransmitStatus(TransmitMailbox)) == CANTXPENDING);

        i++;
        // Tries to send message as many times necessary
    } while ((retval != CANTXOK) && (i <= CAN_RETRIES));

    return retval;
}

```

A megfelelő struktúra kitöltése után megpróbálja elküldeni a keretet. Ha nem tudja bemásolni a TX FIFO³⁴-ba, akkor megpróbálja újra. Akkor is újra próbálkozik, ha sikerül ugyan az üzenetet a elküldeni, de nem érkezik időben a nyugta. Természetesen meg kell határozni, hogy hányszor van értelme elküldeni egy üzenetet: folyamatosan próbálkozva lehet, hogy előbb-utóbb valamelyik vevő válaszol, de addigra az információ esetleg elavul, vagy félrevezető lehet. Ezért alapesetben ez a függvény 16-szor tesz kísérletet:

```

#ifndef CAN_RETRIES
#define CAN_RETRIES 16
#endif

```

A vételi függvényre akkor lesz szükségünk, ha egy sikeresen vett üzenet hatására a CAN vezérlő megszakítást generál. Ekkor a következő módon olvashatjuk ki az üzenetet az RX FIFO-ból:

```

// -----
// Gets the information from the CAN RX FIFO. The return value is a
// pointer to the structure that contains the received and processed data.
// -----

CanRxMsg *CAN_recv() {
    /* receive */
    RxMessage.StdId = 0x00;
    RxMessage.IDE = CAN_ID_EXT;
    RxMessage.DLC = 0;
    RxMessage.RTR = 0;
    RxMessage.Data[0] = 0x00;
    RxMessage.Data[1] = 0x00;
    RxMessage.Data[2] = 0x00;
    RxMessage.Data[3] = 0x00;
    RxMessage.Data[4] = 0x00;
    RxMessage.Data[5] = 0x00;
    RxMessage.Data[6] = 0x00;
    RxMessage.Data[7] = 0x00;

    CAN_Receive(CAN_FIFO0, &RxMessage);

    return &RxMessage;
}

```

Ez a nem túl bonyolult függvény először törli a struktúra tartalmát, majd a megfelelő memóriaterületekről beolvastatja az érkezett üzenet tartalmát a struktúra mezőibe. Ezt a firmware library alacsony szintű függvényén keresztül éri el.

34 Adási tároló

Zitám tud fényképet csinálni szkópról!

3.2.4. A megszakítási rutinok működése

Megszakítás akkor keletkezik, amikor az USART-on keresztül karakter érkezik, vagy a CAN vezérlő üzenetet vesz. Ehhez mindkét eszközt megfelelően inicializálni kell. Erre a `sysinit.c` fájl szolgál:

```
// -----
// The next words should be pointers to ISRs (Interrupt Service Routines).
// These parameters will be placed into the ".vectors" section.
// See also: linker script
// -----

void no_handler();

__attribute__((section(".vectors")))
void (*vectors[])() = {
    sysinit, no_handler, no_handler, no_handler, no_handler,
    no_handler, no_handler, no_handler, no_handler, no_handler,
    no_handler, no_handler, no_handler, no_handler, systick,
    no_handler, no_handler, no_handler, no_handler, no_handler,
    no_handler, no_handler, no_handler, no_handler, no_handler,
    no_handler, no_handler, no_handler, no_handler, no_handler,
    CAN_rx_handler, no_handler, no_handler, no_handler, no_handler,
    no_handler, no_handler, no_handler, no_handler, no_handler,
    no_handler, no_handler, no_handler, no_handler, no_handler,
    no_handler, no_handler, usart_rx_handler, no_handler, no_handler,
};

// -----
// When a non-expected interrupt or exception occurs,
// this routine will "halt" the CPU.
// -----

void no_handler() {
    for(;;);
}

// -----
// Queues to the CAN and USART routines.
// -----

t_queue usart2can;
t_queue can2usart;
const unsigned int f_cpu = CLOCK_FREQ * PLL_FACTOR;

// -----
// The function will be started after RESET.
// -----

void sysinit() {
    // Structure to configure Interrupt Controller
    NVIC_InitTypeDef NVIC_InitStructure;

    // Clean RAM area
    unsigned char *p;

    for (p = (unsigned char *) RAM_BASE;
         p < ((unsigned char *) RAM_BASE + RAM_SIZE); p++)
```

```

    *p = 0;

    // Enable main (Quartz) oscillator
    clock_enable_main_osc();
    // Enable PLL: see also PLL_MUL and PLL_DIV constants
    clock_enable_pll1();

    // Low level initialization of the GPIO ports
    gpio_init();
    // Low level initialization of the USART
    usart_init(USART_BAUDRATE);
    // Low level initialization of the CAN
    CAN_init(f_cpu / 32 / CAN_BAUDRATE - 1);
    CAN_set_filter(0, 0, 0);

    // Initialize global variables (flags and queues)
    cr_needed = line_len = 0;
    queue_init(&usart2can);
    queue_init(&can2usart);

    // Initialization of the SysTick Timer
    // Parameter: period time: 1/n sec, where "n" is the parameter
    systick_init(4);

    // Enable the USART1 Interrupt
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQChannel;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);

    // Enable CAN RX0 interrupt IRQ channel
    NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN_RX0_IRQChannel;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    // CAN FIFO0 and FIFO1 message pending interrupt enable
    CAN_ITConfig(CAN_IT_FMP0, ENABLE);
    CAN_ITConfig(CAN_IT_FMP1, ENABLE);

    // Finally, the main function will be started
    while (1)
        main();
}

```

Az itt látható `sysinit.c` fájl a `vectors` tömb inicializálásával kezdődik: ily módon adjuk meg a megszakítási vektorok (`systick`, `CAN_rx_handler`, `usart_rx_handler`) címét. A `sysinit` függvény néhány alapvető beállítás elvégzése után engedélyezi az oszcillátort és a PLL-t, majd inicializálja a GPIO-t, az USART-ot és a CAN-t. Ezt követően kerül sor a már ismertetett queue-k (üzenetsorok) alapbeállítására.

A `systick` számláló értékének beállítását követi a számunkra legérdekesebb rész: a megszakításkezelő alacsony szintű függvényeivel engedélyezzük az USART és a CAN RX megszakítását. A megfelelő IRQ csatorna (IRQ channel) azonosítója az STM32 dokumentációjában található meg.

CAN esetén szükséges mindkét FIFO vételi megszakításának engedélyezése.

A legutolsó művelet, amit a `sysinit` függvény elvégez az, hogy elindítja a C nyelv szabványos, `main` függvényét.

Az USART és a CAN megszakítási rutinja nagymértékben eltér egymástól. Az előbbi karakterenként fogadja az adatokat, parancsokat, majd szintaktikai és szemantikai ellenőrzés után üzenetté (queue üzenetté) alakítja azokat.

A CAN rutinja egy lépésben kapja meg a keretet, és azt alakítja queue üzenetté.

Az így előálló üzeneteket a főprogram feldolgozza, és szükség esetén adást kezdeményez az USART-on illetve a CAN interfészen keresztül.

Lássuk a megszakításkiszolgáló rutinok felépítését (`irq.c`)!

```
// -----  
// There are two "diagnostic" LEDs used by the program.  
// The first one is to indicate that the state machine started to receive  
// a new command through serial line.  
// The second LED is a blinking one, which shows that the device is working.  
// -----  
  
// -----  
// PORT B 15 (blinking) and 14 (receiving new message).  
// -----  
  
#define LED_SYSTICK (1 << 15)  
#define LED_USART  (1 << 14)  
  
// -----  
// Global and volatile variables: line received from the upper controller.  
// -----  
  
volatile char line[MAX_STR_LEN];  
volatile unsigned int line_len;  
volatile unsigned int cr_needed;
```

Ez a programrész konstansok definiálásával kezdődik: meghatározzuk, hogy a két LED (a systick által működtetett villogó LED, amely a processzor működőképességét jelzi, és az USART vételjelző LED-je) a mikrovezérlő melyik PORTB lábán keresztül érhető el.

Ezt követően deklaráljuk a `line` változót, amely az a sztring, melyben a beérkezett karakterek sorozatát tároljuk. Tesszük ezt mindaddig, amíg soremelés karakter érkezik, mert ekkor a teljes sort fel kell dolgozni, és szükség esetén üzenetet kell küldeni a főprogramnak. A `line_len` értéke a `line` aktuális hosszán tárolja. A `cr_needed` azt jelzi, hogy érkezett-e kocszi vissza (cursor return) karakter a soros porton keresztül. Ha érkezett, akkor a válaszban mi is visszaküldjük, hogy növeljük az operációs rendszerek közötti átjárhatóságot.

```
// -----  
// Message queues to CAN and to USART controlling routines.  
// -----  
  
extern t_queue usart2can;  
extern t_queue can2usart;  
  
// -----  
// Stores the value of the "blinking" LED.  
// -----  
  
volatile unsigned int counter;
```

A két queue szerepe az, hogy üzeneteket küldjenek a megszakítási rutinok a főprogramnak. Ezek itt „külső”, extern változók, tehát egy másik modulban már deklaráltuk ezeket.

A counter értékét, ami egy közönséges számláló, a systick időzítő használja annak jelzésére, hogy a systick LED éppen világít vagy sötét.

Az USART megszakításkiszolgáló rutinja a legösszetettebb minden függvény közül, ezért ennek megismerése csak szakaszokban célszerű.

```
void usart_rx_handler() {
    char ch;
    t_message msg;

    // If a new character is received
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) {
        // Switch on "Receiving..." LED
        gpio_set(LED_USART);

        // Read the character from the USART FIFO
        ch = usart_recv_char();
    }
}
```

Ez a függvény akkor fut le, ha az USART-on keresztül egy karakter érkezik. A függvény (biztos, ami biztos alapon) leellenőrzi, hogy valóban így történt-e. Ha igen, akkor bekapcsolja az USART LED-et a 2.4. fejezetben már ismertetett módon. Ezután kiolvassa a karakter értékét a vételi tárolóból.

```
// A CR (Cursor Return) indicates that CR must be sent after
// the answer string.

if (ch == '\r') {
    cr_needed = 1;
}
```

Ha a vett karakter egy kocszi vissza jel, akkor megjegyezzük, mert a választ is majd ezzel kell zárni. Ha soremelés (line feed) karaktert vettünk, akkor vége a kapott parancsnak, tehát el kell kezdeni feldolgozni az eddig érkezett karaktersorozatot, a tulajdonképpeni sort (a parancsok pontos formátuma a 3.2.6. fejezetben található):

```
// At the end of the command line...
if ((ch == '\n') || (ch == '\r')) {
    // ... when the command starts with...
    switch (line[0]) {
        case 'v':
        case 'V':
            // Version info
            // One character expected
            msg.command = CAN_VERSION;
            msg.param1 = msg.param2 = msg.param3 = 0;
            break;
    }
}
```

Ha a sor első betűje egy „v” vagy „V” betű, akkor a verziószámot kell visszaadni. Mivel a küldés a főprogram kiváltsága, ezért jelezzük neki, hogy küldje el a verzió sztringet a soros porton keresztül. Ezt úgy érjük el, hogy egy olyan üzenetet teszünk az üzenetsorába, melynek parancs mezője a CAN_VERSION³⁵ konstans értékével egyezik meg, a paraméterek értéke pedig nulla.

35 A többi konstans értéke a can_cmd.h-ban található.

```
case 'm':
case 'M':
    // Get free memory
    // Format: M
    msg.command = GET_FREE_MEM;
    msg.param1 = msg.param2 = msg.param3 = 0;
    break;
```

Ez a funkció nagyban hasonlít az előbbihez, de a verziószám helyett a szabad memória közelítő értékét adja vissza.

```
case 'u':
case 'U':
    // Set USART baud rate
    // Format: U_xxxxxxx
    // where 'x' is a hexadecimal digit
    // and '_' is "don't care"
    msg.command = USART_SET_BAUD;
    msg.param1 = hex2num((char *) line+2, 8);
    msg.param2 = msg.param3 = 0;
    break;
```

Ez a szakasz arra szolgál, hogy a vezérlő számítógép programja beállíthassa az USART átviteli sebességét. A parancssor feldolgozása némiképp bonyolultabb, mint az előző esetekben, mert az első betűn kívül paramétert is várunk: az átviteli sebesség értékét hexadecimális alakban. A hexadecimális-számérték konverziót egy függvény (`hex2num`) végzi el³⁶. Az átalakított szám lesz a főprogramnak küldendő üzenet paramétere.

```
case 'b':
case 'B':
    // Set CAN baud rate
    // Format: B_xxxxxxx
    // where 'x' is a hexadecimal digit
    // and '_' is "don't care"
    msg.command = CAN_SET_BAUD;
    msg.param1 = hex2num((char *) line+2, 8);
    msg.param2 = msg.param3 = 0;
    break;
```

Ez a kódrész majdnem teljesen megegyezik az előzővel. A különbség mindössze annyi, hogy ez a CAN busz sebességét módosítja.

```
case 'f':
case 'F':
    // Set filter mask
```

36 A `hex2num` függvény megtalálható a mellékletek közt.

```

// Format: Fxxxxxxxxr or Fxxxxxxxxd
// where 'xxxxxxx' is the mask in hexadecimal format
// 'r' is the RTR bit (can be: "r", "R" or empty)
// 'd' is the DATA bit (can be: "d", "D" or empty)
// 'n' is the filter's number (0 ... 13) in hexadecimal
msg.command = CAN_SET_FILTER;
msg.param1 = hex2num((char *) line+1, 1);
msg.param2 = hex2num((char *) line+2, 8);

if ((line[10] == 'r') || (line[10] == 'R'))
    msg.param2 |= 0x80000000;

msg.param3 = 0;

break;

```

Ez a programrész a CAN vevő szűrőjét állítja be a főprogrammal. Mind a szűrő sorszáma, mind az új értéke paraméterként kell, hogy rendelkezésre álljon. Ha az utolsó karakter egy „r” vagy „R” betű, akkor a szűrőt úgy állítja be, hogy távoli keretekre figyelését is lehetővé tegye.

```

case 'i':
case 'I':
    // Set CAN id
    // Format: Inxxxxxxxxr or Inxxxxxxxxd
    // where 'xxxxxxx' is the ID value in hexadecimal
    // 'r' is the RTR bit (can be: "r", "R" or empty)
    // 'd' is the DATA bit (can be: "d", "D" or empty)
    // 'n' is the filter's number (0 ... 13) in hexadecimal
    msg.command = CAN_SET_ID;
    msg.param1 = hex2num((char *) line+1, 1);
    msg.param2 = hex2num((char *) line+2, 8);

    if ((line[10] == 'r') || (line[10] == 'R'))
        msg.param2 |= 0x80000000;

    msg.param3 = 0;

    break;

```

Ez a kód arra szolgál, hogy a megfelelő szűrőhöz tartozó azonosítót módosítsa. Az előző funkcióhoz hasonlóan ez is fel van készítve az RTR bit kezelésére.

```

case 's':
case 'S':
    // Send CAN frame
    // Format: Sniiiiiiiivvvvvvvvvvvvvvr
    //          or Sniiiiiiiivvvvvvvvvvvvvvd
    // where 'n' is the number of bytes to be sent (1 ... 8)
    //          it is not possible to send 0 byte of data!
    // 'iiiiiii' is the destination id in HEXADECEMAL
    // 'vv ... vv' is the data. 16 digits (8 bytes) expected.
    // 'r' the RTR bit (can be: "r", "R" or empty)
    // 'd' is the DATA bit (can be: "d", "D" or empty)
    msg.command = CAN_SEND | (hex2num((char *) line+1, 1) << 16);

```

```
msg.param1 = hex2num((char *) line+2, 8) & 0xffffffff;
msg.param2 = hex2num((char *) line+10, 8);
msg.param3 = hex2num((char *) line+18, 8);

if ((line[26] == 'r') || (line[26] == 'R'))
    msg.param1 |= 0x80000000;

break;
```

Ha az „s” vagy „S” betűt megfelelő paraméterek követik, akkor a mikrovezérlő CAN keretet küld a beállításoknak és a paramétereknek megfelelően. Meg kell adnunk azt, hogy hány adatbájtot szeretnénk küldeni, milyen üzenet azonosítóval, és pontosan melyek legyenek a küldendő adatok. Mindezt hexadecimális formában tudjuk megmondani. Ha az utolsó betű „r” vagy „R”, akkor a vezérlő beállítja az RTR bitet, tehát a keretünk egy remote frame lesz.

```
default:
    // When received an invalid command
    msg.command = CAN_UNKNOWN;
    msg.param1 = msg.param2 = msg.param3 = 0;
    break;
}
```

Ha a karaktersorozat nem az előbb felsorolt betűkkel kezdődik, akkor szintaktikailag hibás a parancssor. Ekkor egy hibajelzést küldünk a főprogramnak.

Végezetül el kell küldeni a kitöltött struktúrának megfelelően az üzenetet, amely a már megismert módon (queue kezelő függvények) lehetséges. A parancssor feldolgozása után a LED-et ki kell kapcsolni, így a parancssor átvitele során világít, míg a parancs végrehajtása után kialszik.

```
// Sending the CAN command to the routine that
// utilizes CAN controller
queue_put_non_blocking(&usart2can, &msg);
// Switch off the "Receiving..." LED
// at the end of the command line
gpio_clear(LED_USART);
// Line is to be invalidated
line_len = 0;
line[0] = 0;
return;
```

Ha a adat nem soremelés vagy újsor karakter, akkor hozzá kell fűzni a már eddig is vett adatokhoz. Így egy teljes értékű karakterfűzért kapunk. Ez dolgozzuk fel az előbbieken bemutatott módon.

```
} else {
    // If one more character can be attached to the string
    // (command line)...
```

```

        if (line_len + 1 < MAX_STR_LEN) {
            // ... attach the character, and the terminator "zero"
            line[line_len++] = ch;
            line[line_len] = 0;
        }
    }
}

```

A CAN megszakítási rutinja ettől jóval egyszerűbb, mert az nem végez semmilyen szintaktikai vagy szemantikai ellenőrzést, a kapott üzenettel érkező adatokat elhelyezi egy struktúrában, melyet elküld a főprogramnak. A főprogram az adatokat eljuttatja felhasználni számítógépére a soros porton keresztül.

```

// -----
// When a message (frame) has been received through CAN controller...
// -----

void CAN_rx_handler() {
    t_message msg;
    CanRxMsg RxMessage;

    // Clean invalid data from the structure
    RxMessage.StdId = 0x00;
    RxMessage.ExtId = 0x00;
    RxMessage.IDE = 0;
    RxMessage.DLC = 0;
    RxMessage.FMI = 0;
    RxMessage.Data[0] = 0x00;
    RxMessage.Data[1] = 0x00;
    RxMessage.Data[2] = 0x00;
    RxMessage.Data[3] = 0x00;
    RxMessage.Data[4] = 0x00;
    RxMessage.Data[5] = 0x00;
    RxMessage.Data[6] = 0x00;
    RxMessage.Data[7] = 0x00;

    // Copy data from CAN FIFO to structure
    CAN_Receive(CAN_FIFO0, &RxMessage);

    // If it is in the acceptable format (with 29 bit addresses)...
    if (RxMessage.IDE == CAN_ID_EXT) {
        // Preparing the message to be sent to "main()" function
        msg.command = RxMessage.DLC | (RxMessage.RTR << 16);
        msg.param1 = RxMessage.ExtId;
        msg.param2 = (RxMessage.Data[0] << 24) | (RxMessage.Data[1] << 16) |
            (RxMessage.Data[2] << 8) | RxMessage.Data[3];
        msg.param3 = (RxMessage.Data[4] << 24) | (RxMessage.Data[5] << 16) |
            (RxMessage.Data[6] << 8) | RxMessage.Data[7];

        // Send message to "main()", which will send it to "upper"
        // controller
        queue_put_non_blocking(&can2usart, &msg);
    }
}

```

Ezzel a megszakítási rutinok tárgyalását ezzel be is fejeztük. A következő részben az imént ismertetett kódokhoz szorosan kapcsolódó főprogram megismerésével folytatjuk.

3.2.5. A főprogram funkciói

A főprogram legfontosabb feladata, hogy - miután már lezajlott a hardver összetevők alaphelyzetbe állítása - fogadja a megszakítási rutinoktól érkező üzeneteket, és valamiképpen reagáljon azokra. Természetesen a `main.c` fájl is a globális változók deklarálásával kezdődik:

```
// -----  
// Global variables: queues to the CAN and USART routines.  
// -----  
  
volatile t_queue usart2can;  
volatile t_queue can2usart;  
  
// -----  
// These variables contain the CAN baudrate and the filter/mask settings.  
// -----  
  
volatile unsigned int can_baudrate;  
volatile unsigned int usart_baudrate = USART_BAUDRATE;  
volatile unsigned int can_filter[14], can_id[14];
```

Legelőször a két üzenetsornak készítünk helyet a memóriában. Ezek azért globális változók, hogy a többi programrész (így a megszakítási rutin is) is elérhesse őket.

Ezt követi az olyan paraméterek deklarálása, melyek nem konstans értékűek, hanem a program futása során megváltozhatnak: a különböző sebességértékek, és a CAN vezérlő szűrőinek, maszkjainak értéke.

A `main` függvény néhány inicializálási művelet után hozzákezd az üzenetek fogadásához és feldolgozásához:

```
// -----  
// The "main()" function is started by the "sysinit()" routine.  
// The "sysinit()" is the function that is started when the MCU starts.  
// -----  
  
int main() {  
    // A string to store text message to the "upper" CPU or user  
    char str[MAX_STR_LEN];  
  
    // The amount of the free RAM  
    unsigned int free_mem = 1;  
  
    // Pointer to a message in the message queue  
    t_message *pmsg;  
  
    // Predefine the value of the CAN baud rate  
    can_baudrate = CAN_BAUDRATE;  
  
    // Until I realize nirvana...  
    while (1) {  
        // Is there a new message in the USART->CAN queue?  
        // The message arrives from the USART ISR.  
        if (queue_get_non_blocking((t_queue *) &usart2can, &pmsg) == QUEUE_OK) {  
            switch (pmsg->command & 0xffff) {
```

Ha az üzenetsorból érvényes üzenet olvasható ki, akkor a főprogram elkezd a parancs és a paraméterek feldolgozását:

```
case USART_SET_BAUD:
    // Predefined baud rate shall be changed
    usart_baudrate = pmsg->param1;
    // Set the BAUD rate
    usart_init(usart_baudrate);

    // Human-readable string...
    usart_send_str("+ USART Baud rate: 0x");
    // + the numeric of baud rate converted to hexadecimal format
    num2hex(usart_baudrate, str, 8);
    // Send hexadecimal "string"
    usart_send_str(str);
    usart_send_str(".");

    break;
```

Ebben az esetben arra utasít a felhasználó, hogy állítsuk át az USART sebességét. Ezt úgy érjük el, hogy a paraméternek megfelelően újrainicializáljuk a soros portot, majd egy nyugtázó üzenetet küldünk a felhasználónak. Ezt ő csak akkor látja, ha pár mikroszekundumon belül ő is elvégzi a módosítást. Ezt persze egy ember csak ritkán tudja elérni, de egy gyors felhasználói szoftvernek ez nem okoz problémát.

Itt hívnám fel a figyelmet a `num2hex` függvényre, amely a `hex2num` függvény párja, annyi különbséggel, hogy az előbbi egy számot (ebben az esetben a bitsebességet) alakítja „olvasható”, hexadecimális sztringgé.

Az `usart_send_str` függvénnyel már találkoztunk a 2.4. fejezetben.

```
case CAN_VERSION:
    // Get version information:
    // A text message will be sent:
    // constant text + VERSION string
    // (which can be seen in "config.h")
    usart_send_str("+ SER_CAN: ");
    usart_send_str(DEV_VERSION);
    break;
```

Ha a felhasználó a soros-CAN átalakító verziószámára kíváncsi, akkor a „+SER_CAN: ” kezdetű, és a `DEV_VERSION` sztringgel folytatódó karaktersorozatot kapja válaszul. A `DEV_VERSION` értéke a `config.h`-ban található. Jelenleg a következő értéket tartalmazza: „Fuszenecker's serial-CAN converter, Release IV.”

```
case GET_FREE_MEM:
    // Get free memory
    usart_send_str("+ Free memory: 0x");
    // Convert the number to hexadecimal format
    free_mem = get_free_mem();
    num2hex(free_mem, str, 8);
    // Print the amount of free memory
    usart_send_str(str);
```

```
// Memory in use
usart_send_str(" bytes, memory in use: 0x");
// Convert the number to hexadecimal format
num2hex(RAM_SIZE - free_mem, str, 8);
// Print the amount of used memory
usart_send_str(str);
usart_send_str(" bytes.");

break;
```

Ha a szabad és a foglalt memória megállapítása a feladat, akkor meg kell állapítani azt, hogy a memória mely részei szabadok. Erre külön függvény szolgál: a `get_free_mem()`.

A visszatérési értéke azon területek méretének összege, melyek csak nulla értéket tartalmaznak. Működését később ismerhetjük meg részletesebben.

Ez a kódrészlet nem csak a szabad memória méretét küldi el a felhasználónak, hanem a használt memóriaterületek méretét is. A jelenlegi verzió kb. 1 kilobájt RAM-ot használ a globális változók tárolására, és ugyanennyit foglal a veremterület is.

```
case CAN_SET_BAUD:
    // Predefined baud rate shall be changed
    can_baudrate = pmsg->param1;
    // See equation above
    CAN_set_prescaler(f_cpu / 32 / can_baudrate - 1);

    // Human-readable string...
    usart_send_str("+ CAN Baud rate: 0x");
    // + the numeric of baud rate converted to hexadecimal format
    num2hex(can_baudrate, str, 8);
    // Send hexadecimal "string"
    usart_send_str(str);

    usart_send_str(" , divisor: 0x");
    num2hex(f_cpu / 32 / can_baudrate - 1, str, 8);
    usart_send_str(str);
    usart_send_str(".");
    break;
```

Abban az esetben, ha a felhasználó szeretné megváltoztatni a CAN busz sebességét, akkor újra kell számolni a baudrate generátor előosztójának osztási arányát (`prescaler`), és az alacsony szintű függvények segítségével meg kell változtatni a CAN vezérlő paramétereit. A felhasználót mindenképpen értesítjük a művelet elvégzése után.

```
case CAN_SET_FILTER:
    // Set filter mask value:
    // The filter identifier must be less than 14!
    if ((pmsg->param1 < 14) && (pmsg->param1 >= 0)) {
        // Save mask value to the array defined above
        can_filter[pmsg->param1] = pmsg->param2 & 0x9fffffff;
        // Set filter according to the saved value
        CAN_set_filter(pmsg->param1, can_filter[pmsg->param1],
```



```

        can_id[pmsg->param1]);

    // A little string to the user:
    // The values will be sent back
    usart_send_str("+ Filter 0x");
    num2hex(pmsg->param1, str, 1);
    usart_send_str(str);

    usart_send_str(": Mask: 0x");
    num2hex(can_filter[pmsg->param1], str, 8);
    usart_send_str(str);

    usart_send_str(", Id: 0x");
    num2hex(can_id[pmsg->param1], str, 8);
    usart_send_str(str);

    if (can_filter[pmsg->param1] & 0x80000000)
        usart_send_str(", RTR");

    usart_send_str(".");
} else {
    // if the filter ID >= 14 or ID < 0, then an
    // error message should be sent
    usart_send_str("- Invalid filter number.");
}

break;

case CAN_SET_ID:
    // Set ID value:
    // The filter identifier must be less than 14!
    if ((pmsg->param1 < 14) && (pmsg->param1 >= 0)) {
        // Save filter ID value to the array defined above
        can_id[pmsg->param1] = pmsg->param2 & 0x9fffffff;
        // Set filter according to the saved value
        CAN_set_filter(0, can_filter[pmsg->param1],
            can_id[pmsg->param1]);

        // A little string to the user:
        // The values will be sent back
        usart_send_str("+ Filter 0x");
        num2hex(pmsg->param1, str, 1);
        usart_send_str(str);

        usart_send_str(": Mask: 0x");
        num2hex(can_filter[pmsg->param1], str, 8);
        usart_send_str(str);

        usart_send_str(", Id: 0x");
        num2hex(can_id[pmsg->param1], str, 8);
        usart_send_str(str);

        if (can_filter[pmsg->param1] & 0x80000000)
            usart_send_str(", RTR");

        usart_send_str(".");
    } else {
        // if the filter ID >= 14 or ID < 0, then an
        // error message should be sent
        usart_send_str("- Invalid filter number.");
    }

    break;

```

A szűrő paraméterek (maszk és azonosító) beállítása úgy történik, hogy ellenőrizzük, hogy a megadott szűrőszám alkalmas-e. Ha nem az, akkor hibajelzést küldünk a felhasználónak. Minden szűrőparamétert tömbben tárolunk. Először ezt a tömböt módosítjuk, majd ennek alapján frissítjük a CAN vezérlő beállításait.

A művelet elvégzése után tájékoztatjuk a felhasználót.

```
case CAN_SEND:
    // Send message through CAN bus:
    // Try to send the message, on success...
    if (CAN_send(pmsg->param1 & 0x1fffffff,
                pmsg->command >> 16,
                pmsg->param2, pmsg->param3,
                pmsg->param1 & 0x80000000) == CANTXOK) {
        // ... a positive acknowledge is sent ...
        usart_send_str("+ Message successfully sent.");
    } else {
        // ... else a warning will be received on USART
        usart_send_str("- Unable to send message.");
    }
    break;
```

A CAN üzenet küldését végző kódrészlet meglehetősen egyszerű. Ennek az az oka, hogy a `CAN_send` függvénybe került a műveletek jelentős része. Tulajdonképpen nem teszünk mást, mint meghívjuk a kapott paraméterekkel a `CAN_send` függvényt, és ellenőrizzük, hogy sikerült-e elküldeni a CAN keretet. Akár sikerült, akár nem, a megfelelő üzenettel értesítjük a felhasználót.

```
case CAN_UNKNOWN:
    // When an unknown CAN message arrives, a warning should appear
    usart_send_str("- Invalid command.");
    break;
```

Fel kell készülnünk arra is, hogy nem mindig kapunk megfelelő parancsot a megszakításkiszolgáló rutintól. Ez nem túl valószínű, de nem is lehetetlen. Ha mégis megtörténne, egy hibaüzenetet kap a felhasználó.

```
// After sending the message(s), a newline should be sent, as well.
// But what kind of newline? CR+LF or LF only?
// It depends on the "upper" device or on the user.
if (cr_needed)
    usart_send_str("\r\n");
else
    usart_send_str("\n");
```

Ha a parancsot küldő fél kocsit vissza³⁷ karakterrel zárta a sort, akkor mi is így teszünk.

³⁷ Cursor Return – kocsit vissza; I. ASCII tábla

```

// The received and processed message should be removed from
// the queue, so that some space appears in the queue.
queue_remove((t_queue *) &usart2can);
}

```

Ha végrehajtottuk a parancsot, akkor célszerű kivenni a várakozási sorból, hogy ezzel helyet biztosítsunk újabb parancsok és paramétereik számára.

A főprogram további része akkor kerül előtérbe, amikor CAN keretek érkeznek a buszon keresztül, és az aktivált megszakításrutin üzenetet küld erről:

```

// If there is a message in the CAN queue arrived from the CAN ISR...
if (queue_get_non_blocking((t_queue *)&can2usart, &pmsg) == QUEUE_OK) {
    // Provide a string that contains the ID, ...
    usart_send_str("# id: 0x");
    num2hex(pmsg->param1 & 0xffffffff, str, 8);
    usart_send_str(str);

    // ... length ...
    usart_send_str(", length: ");
    num2hex(pmsg->command & 0xffff, str, 1);
    usart_send_str(str);

    // ... and data field.
    usart_send_str(", data: 0x");
    num2hex(pmsg->param2, str, 8);
    usart_send_str(str);
    num2hex(pmsg->param3, str, 8);
    usart_send_str(str);

    // Remote frame?
    if (pmsg->command >> 16)
        usart_send_str(", RTR");

    usart_send_str(".");

    // The string will be terminated as described above.
    if (cr_needed)
        usart_send_str("\r\n");
    else
        usart_send_str("\n");

    // The received and processed message should be removed from
    // the queue, so that some space appears in the queue.
    queue_remove((t_queue *) &can2usart);
}
} /* while (1) ... */
}

```

Ha van új üzenet a queue-ban, akkor elkérjük annak címét, és a kapott paraméterek alapján egy meglehetősen hosszú, de könnyen érthető üzenetet küldünk a felhasználónak az USART-on keresztül. Természetesen ez a kódrészlet is fel van készítve a távoli keretek fogadására. A művelet végén töröljük a beérkezett üzenetet.

A főprogram legvégén a szabad memória méretét megállapító függvény kapott helyet. Ez a kódrészlet nem csinál mást, mint logikailag lapokra osztja a memóriaterületeket, és megszámlolja, hogy hány olyan lap van, amely csak nulla értéket tartalmaz. A lapok méretének összege közelítőleg a szabad memória méretét adja meg.

```
// -----  
// The get_free_mem() function determines the free memory (RAM).  
// -----  
  
unsigned int get_free_mem() {  
    unsigned char *p = (unsigned char *) RAM_BASE;  
    unsigned int i, flag, free = 0;  
  
    for (; p < ((unsigned char *) RAM_BASE + RAM_SIZE); p += PAGE_SIZE) {  
        for (flag = 1, i = 0; i < PAGE_SIZE; i++)  
            if (p[i] != 0)  
                flag = 0;  
  
        if (flag)  
            free++;  
    }  
  
    return free * PAGE_SIZE;  
}
```

Ez persze feltételezi, hogy kezdetben a teljes memória nulla értékekkel volt feltöltve, és a függvények adatokat helyeztek/helyeznek a felhasznált területekre. Kezdetben tehát gondoskodni kell a memória törléséről (sysinit.c):

```
// Clean RAM area  
unsigned char *p;  
  
for (p = (unsigned char *) RAM_BASE;  
     p < ((unsigned char *) RAM_BASE + RAM_SIZE); p++)  
    *p = 0;
```

Ezzel végeztünk a teljes szoftverhátér ismertetésével. A következő részben már a lefordított, és működő kódot fogjuk megvizsgálni, elsősorban a kommunikáció szempontjából, vagyis melyik interfészen hogyan történik az adatok, üzenetek továbbítása, értelmezése.

3.2.6. Kommunikáció a PC-vel: CLI és GUI

Ebben a részben arra kapunk választ, hogy hogyan tudunk kapcsolatot teremteni a felhasználó számítógépe és a soros-CAN átalakító között. Először egy terminál emulátor program (minicom) segítségével áttekintjük az átalakítónak kiadható parancsokat, azok formátumát, paramétereit, majd megnézzük azt is, hogy a megszerzett ismeretek alapján hogyan tudunk grafikus felhasználói felületet készíteni Python nyelven.

A minicom indítása után be kell állítanunk a soros port sebességét az átalakító alapértelmezett beállításai alapján. Ez 1200 bit/s-ot jelent, 8 adatbittel, paritásvédelem nélkül, 1 stop bit felhasználásával.

Ha elkezdjük begépelni a parancsokat, az USART LED világítani kezd, jelezve, hogy sikerült venni a leütött karaktert. Ez persze nem jelenti azt, hogy a vevő oldal azt vette, amit küldtünk neki. Ha a parancs begépelése

után leütjük az ENTER billentyűt, és a LED elalszik, akkor a parancs megérkezett az átalakítóhoz. Ha várunk még pár 100 ms-ot, akkor a válasz is fog érkezni a parancsra.

Ahogy már többször is ígértem, ebben a részben pontosan összefoglalom a használható parancsokat, azok formátumát és paramétereit. A parancsok nem érzékenyek a kis- és nagybetűkre, ezért a következő lista csak az egyik lehetőséget tartalmazza. Minden számot **hexadecimális alakban** kell megadni, és a válaszban is ilyen formában kapjuk vissza a számértékeket.

- A verzióstring lekérdezése:
 - Formátuma: **v**
 - Példa: **„v”**
 - Lehetséges válasz:

„+ SER_CAN: Fuszenecker's serial-CAN converter, Release IV.”

- Szabad (és felhasznált) memória méretének lekérdezése:
 - Formátuma: **m**
 - Példa: **„m”**
 - Lehetséges válasz:

„+ Free memory: 0x00001C00 bytes, memory in use: 0x00000400 bytes.”

- Soros kommunikáció sebességének beállítása (bbbbbbbb hexadecimális értékre bit/s mértékegységben; a „_” helyén bármi állhat, kivéve a soremelés karakter):
 - Formátuma: **u_bbbbbbbb**
 - Példa: **„u_0001c200”**
 - Lehetséges válasz:

„+ USART Baud rate: 0x0001C200.”

- CAN busz sebességének beállítása (ssssssss hexadecimális értékre bit/s egységben; a „_” helyén bármi állhat, kivéve a soremelés karakter):
 - Formátuma: **b_ssssssss**
 - Példa: **„b_0001e848”**
 - Lehetséges válasz:

„+ CAN Baud rate: 0x0001E848, divisor: 0x00000011.”

- CAN maszk beállítása (n a szűrő sorszáma, xxxxxxxx a maszk új értéke, „r” pedig azt jelzi, hogy a szűrő érzékeny legyen-e az RTR bitre):
 - Formátuma: **fn_xxxxxxxr**
 - Példa: **„f30000ff00r”** vagy **„f21000ffff”**
 - Lehetséges válasz:

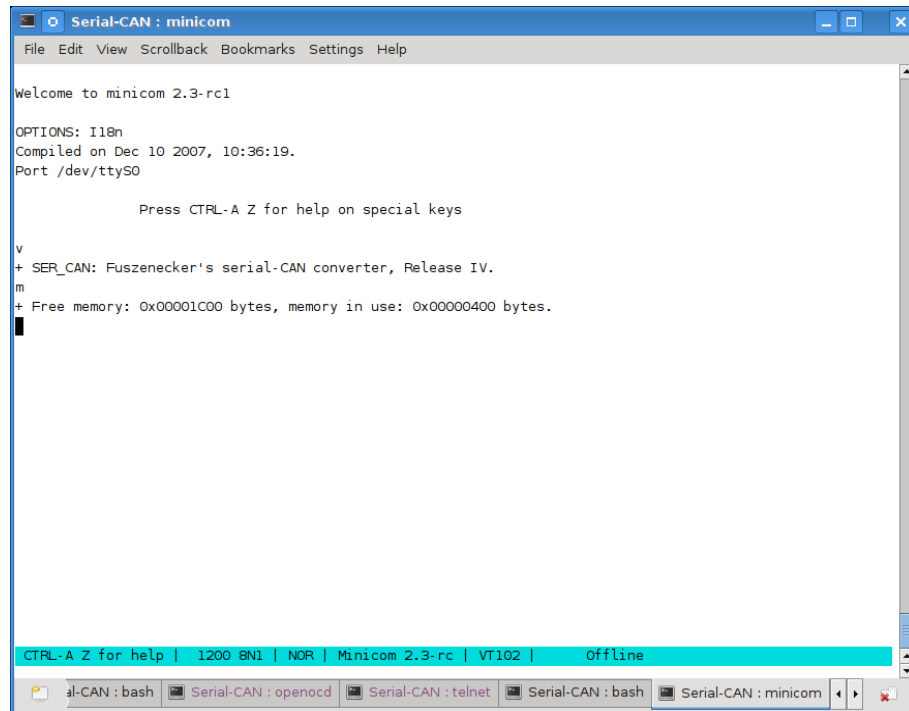
„+ Filter 0x3: Mask: 0x8000FF00, Id: 0x00000000, RTR.” vagy
 „+ Filter 0x2: Mask: 0x1000FFFF, Id: 0x00000000.”

- CAN szűrő azonosító paraméterének beállítása (n a szűrő sorszáma, xxxxxxxx az azonosító új értéke, „r” pedig azt jelzi, hogy a szűrő akkor „engedi be” a CAN üzenetet, ha az RTR bitje is „1” értékű):
 - Formátuma: **in_xxxxxxxr**
 - Példa: **„i307ff55aar”** vagy **„i21000ffff”**
 - Lehetséges válasz:
„+ Filter 0x3: Mask: 0x8000FF00, Id: 0x87FF55AA, RTR.” vagy
„+ Filter 0x2: Mask: 0x1000FFFF, Id: 0x1000FFFF.”

- CAN üzenet küldése (n az adatbájtok száma, iiiiiiii az azonosító értéke, dddddddddddddddd az adatbájtokat jelenti, „r” pedig azt jelzi, hogy az RTR bit „1” értékű legyen-e a kimenő üzenetben):
 - Formátuma: **sniiiiiiiiddddddddddddddddr**
 - Példa: **„s8100055aaDeadBeefDeadBeefR”** vagy **„s31fff00001122334455667788”**
 - Lehetséges válasz:
„+ Message successfully sent.” vagy
„- Unable to send message.”

- CAN keret vételekor a következő formátumú üzenetet láthatjuk a terminál emulátorban:
„# id: 0x100055AA, length: 8, data: 0xDEADBEEFDEADBEEF, RTR.” vagy
„# id: 0x1FFF0000, length: 3, data: 0x1122330000000000.”

A következő néhány képernyőkép a parancsok használatát mutatja be a gyakorlatban. Látható, hogy a soros-CAN átalakító meglehetősen rugalmasan használható.



```
Serial-CAN : minicom
File Edit View Scrollback Bookmarks Settings Help

Welcome to minicom 2.3-rc1

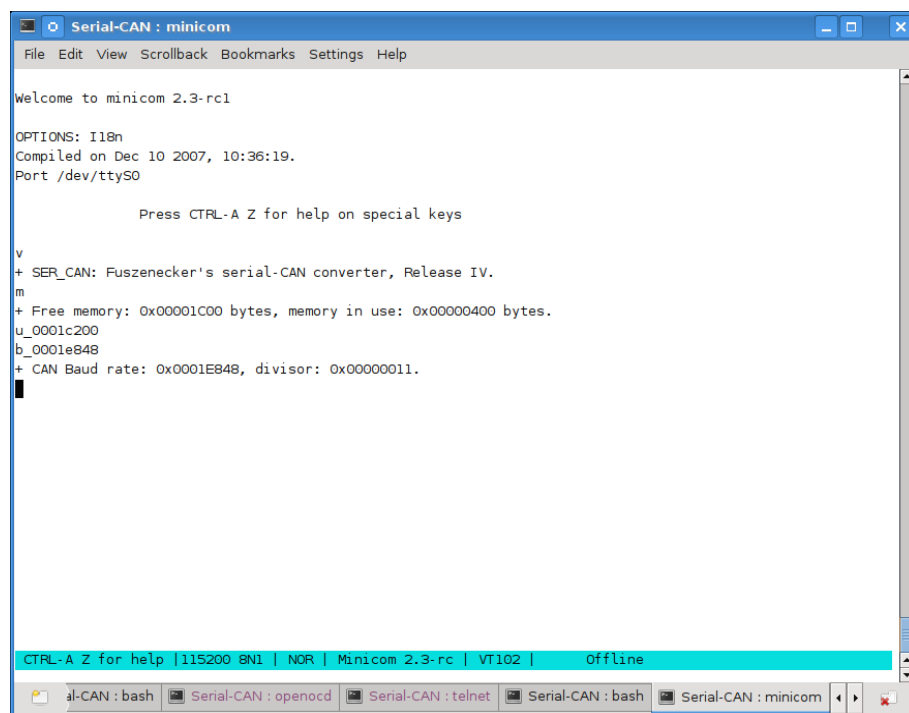
OPTIONS: I18n
Compiled on Dec 10 2007, 10:36:19.
Port /dev/ttyS0

Press CTRL-A Z for help on special keys

v
+ SER_CAN: Fuszenecker's serial-CAN converter, Release IV.
m
+ Free memory: 0x00001C00 bytes, memory in use: 0x00000400 bytes.
█

CTRL-A Z for help | 1200 8N1 | NOR | Minicom 2.3-rc | VT102 | Offline
al-CAN : bash Serial-CAN : openocd Serial-CAN : telnet Serial-CAN : bash Serial-CAN : minicom
```

A képen a verziósztring és a szabad memória lekérdezését láthatjuk.



```
Serial-CAN : minicom
File Edit View Scrollback Bookmarks Settings Help

Welcome to minicom 2.3-rc1

OPTIONS: I18n
Compiled on Dec 10 2007, 10:36:19.
Port /dev/ttyS0

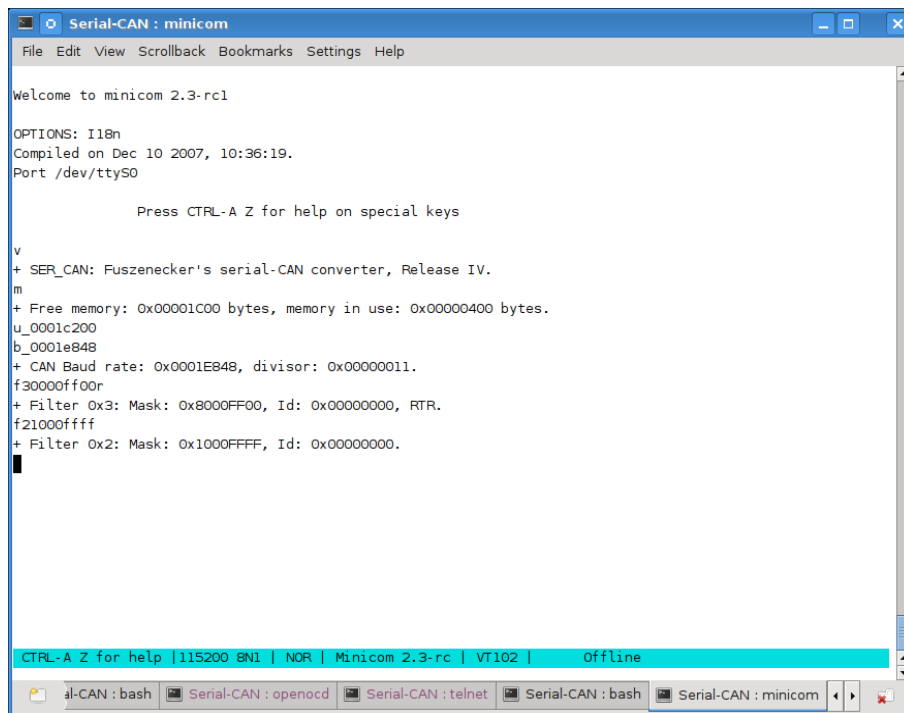
Press CTRL-A Z for help on special keys

v
+ SER_CAN: Fuszenecker's serial-CAN converter, Release IV.
m
+ Free memory: 0x00001C00 bytes, memory in use: 0x00000400 bytes.
u_0001c200
b_0001e848
+ CAN Baud rate: 0x0001E848, divisor: 0x00000011.
█

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.3-rc | VT102 | Offline
al-CAN : bash Serial-CAN : openocd Serial-CAN : telnet Serial-CAN : bash Serial-CAN : minicom
```

Az USART sebességének beállítására adott válasz nem látható, mert nem tudjuk megfelelően gyorsan átállítani a minicom sebességét. A CAN bitráta beállítása hasonlóképpen történik.

3. Soros-CAN átalakító



```
Serial-CAN : minicom
File Edit View Scrollback Bookmarks Settings Help

Welcome to minicom 2.3-rc1

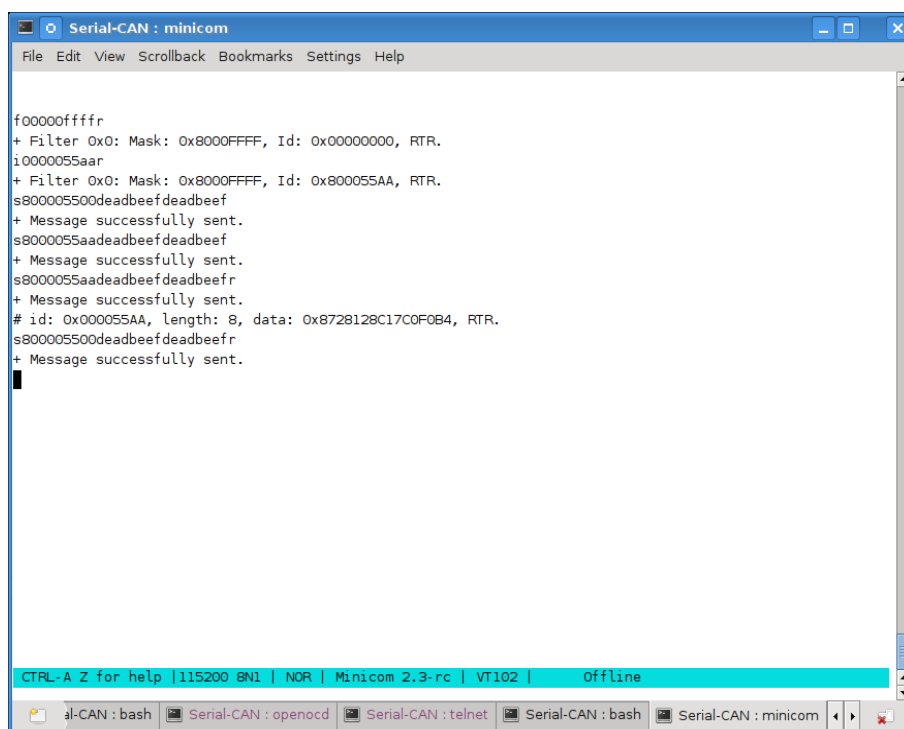
OPTIONS: I18n
Compiled on Dec 10 2007, 10:36:19.
Port /dev/ttyS0

Press CTRL-A Z for help on special keys

v
+ SER_CAN: Fuszenecker's serial-CAN converter, Release IV.
m
+ Free memory: 0x00001C00 bytes, memory in use: 0x00000400 bytes.
u_0001c200
b_0001e848
+ CAN Baud rate: 0x0001E848, divisor: 0x00000011.
f30000ff00r
+ Filter 0x3: Mask: 0x8000FF00, Id: 0x00000000, RTR.
f21000ffff
+ Filter 0x2: Mask: 0x1000FFFF, Id: 0x00000000.
█

CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.3-rc | VT102 | Offline
al-CAN : bash Serial-CAN : openocd Serial-CAN : telnet Serial-CAN : bash Serial-CAN : minicom
```

A képen a szűrők maszkjának és azonosítójának beállítását láthatjuk. Mind az adatkeret, mind a távoli keret fogadása lehetséges.



```
Serial-CAN : minicom
File Edit View Scrollback Bookmarks Settings Help

f00000ffffr
+ Filter 0x0: Mask: 0x8000FFFF, Id: 0x00000000, RTR.
i0000055aar
+ Filter 0x0: Mask: 0x8000FFFF, Id: 0x800055AA, RTR.
sB00005500deadbeefdeadbeef
+ Message successfully sent.
sB000055aadeadbeefdeadbeef
+ Message successfully sent.
sB000055aadeadbeefdeadbeefr
+ Message successfully sent.
# id: 0x000055AA, length: 8, data: 0x8728128C17C0F0B4, RTR.
sB00005500deadbeefdeadbeefr
+ Message successfully sent.
█

CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.3-rc | VT102 | Offline
al-CAN : bash Serial-CAN : openocd Serial-CAN : telnet Serial-CAN : bash Serial-CAN : minicom
```

A képen azt láthatjuk, hogy milyen szűrőbeállítások esetén fogadja a CAN illesztő a kereteket.


```

Serial-CAN : minicom
File Edit View Scrollback Bookmarks Settings Help

f0000ffff
+ Filter 0x0: Mask: 0x8000FFFF, Id: 0x00000000, RTR.
i0000055a
+ Filter 0x0: Mask: 0x8000FFFF, Id: 0x800055AA, RTR.
s800005500deadbeefdeadbeef
+ Message successfully sent.
s8000055aadeadbeefdeadbeef
+ Message successfully sent.
s8000055aadeadbeefdeadbeefr
+ Message successfully sent.
# id: 0x000055AA, length: 8, data: 0x8728128C17C0F0B4, RTR.
s800005500deadbeefdeadbeefr
+ Message successfully sent.

f10000ffff
+ Filter 0x1: Mask: 0x0000FFFF, Id: 0x00000000.
i100001122
+ Filter 0x1: Mask: 0x0000FFFF, Id: 0x00001122.
s3000011001122334455667788
+ Message successfully sent.
s3000011221122334455667788
+ Message successfully sent.
# id: 0x00001122, length: 3, data: 0x1122330000000000.
s3000011221122334455667788r
+ Message successfully sent.
# id: 0x00001122, length: 3, data: 0x824E83A8FA859C19, RTR.

CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.3-rc | VT102 | Offline
al-CAN : bash Serial-CAN : openocd Serial-CAN : telnet Serial-CAN : bash Serial-CAN : minicom

```

Az előző képhez hasonlóan itt is kereteket fogadunk, de ez a beállítás nem érzékeny a távoli keret bit (RTR) értékére.

```

Serial-CAN : minicom
File Edit View Scrollback Bookmarks Settings Help

s800005500deadbeefdeadbeefr
+ Message successfully sent.

f10000ffff
+ Filter 0x1: Mask: 0x0000FFFF, Id: 0x00000000.
i100001122
+ Filter 0x1: Mask: 0x0000FFFF, Id: 0x00001122.
s3000011001122334455667788
+ Message successfully sent.
s3000011221122334455667788
+ Message successfully sent.
# id: 0x00001122, length: 3, data: 0x1122330000000000.
s3000011221122334455667788r
+ Message successfully sent.
# id: 0x00001122, length: 3, data: 0x824E83A8FA859C19, RTR.

f20000ffff
+ Filter 0x2: Mask: 0x8000FFFF, Id: 0x1000FFFF, RTR.
i200002222
+ Filter 0x2: Mask: 0x8000FFFF, Id: 0x00002222, RTR.
s5000011117777777777777777
+ Message successfully sent.
s5000022227777777777777777r
+ Message successfully sent.
s5000022227777777777777777
+ Message successfully sent.
# id: 0x00002222, length: 5, data: 0x7777777770000000.

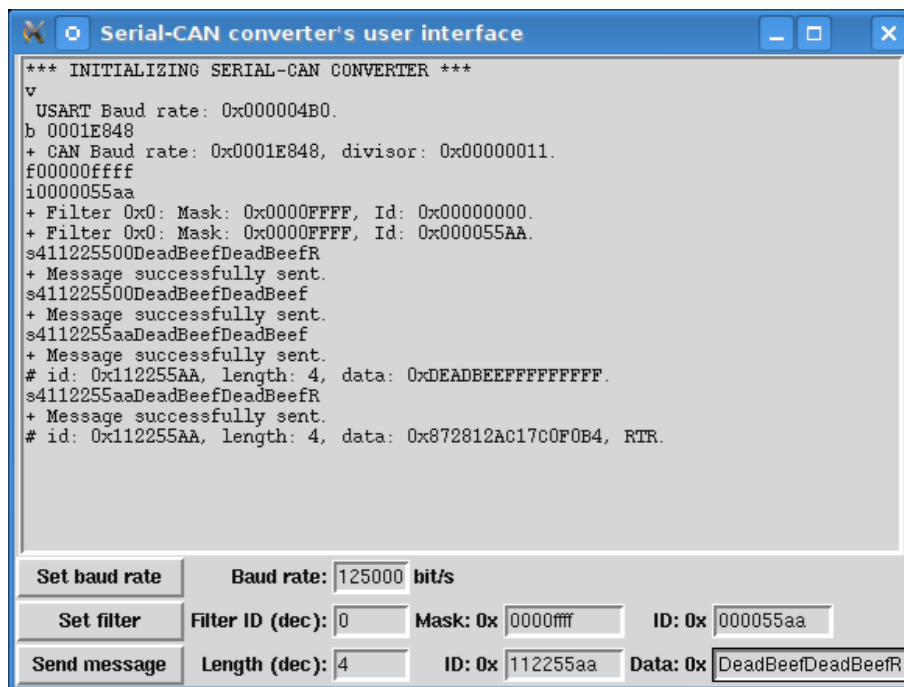
CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.3-rc | VT102 | Offline
al-CAN : bash Serial-CAN : openocd Serial-CAN : telnet Serial-CAN : bash Serial-CAN : minicom

```

A szűrő csak az adatkeretek vételét engedélyezi.

Ha nem terminál emulátor segítségével szeretnénk használni a soros-CAN átalakítót, akkor szükségünk lesz egy olyan programozási nyelvre, amely támogatja a soros port kezelését és a grafikus felhasználói felület létrehozását. A Python³⁸ nyelv mindkét követelménynek megfelel.

Sajnos a Python nyelv ismeretét nem tételezhetem fel, ezért a mintakódot csak érdekességként közlöm a mellékletek közt. Ha az Olvasó kedvet érez hozzá, akkor elmélyülhet a grafikus felület létrehozásának művészetében. A következő kép a grafikus felületet mutatja működés közben:



Egy mivel a Python nyelv platformfüggetlen, azért ez a felhasználói felület nem csak Linux alatt működik, hanem más (MáS) operációs rendszerek alatt is.

³⁸ <http://www.python.org>

4. A DC/DC konverter kialakítása

A dolgozat legfontosabb témaköréhez érkeztünk. Ebben a részben azt fogjuk megnézni, hogy a specifikáció alapján milyen felépítésű DC/DC konvertert érdemes kialakítani, illetve hogyan történik a az alkatrészek (elsősorban a transzformátor) méretezése.

A részletes specifikáció (ld. 1. fejezet) alapján alapján célszerű a szuperkapacitásokat egy áramhatárolt feszültséggenerátorral tölteni. Mivel a követelmények elég nagy teljesítményt írnak elő, ezért mindenképpen kapcsoló üzemű konvertert kell építeni. A megfelelő architektúrát számos tényező figyelembevétele után a 4.1.1. és a 4.1.2. fejezetben fogjuk kiválasztani.

A kapcsolási rajz kialakítása során figyelembe kell venni, hogy a konvertert egy küldetéskritikus eszközben (repülőgép) fogják használni, így a megbízhatóságot maximálisan szem előtt kell tartani.

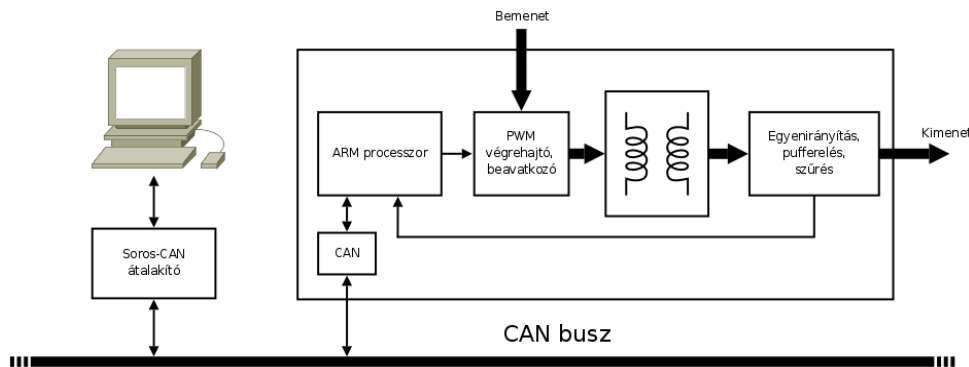
A szoftver keretrendszer kialakítása hasonló gondosságot igényel, mint a hardver megépítése. Ebben a részben jelentősen fogok támaszkodni a 2.3. fejezetben leírtakra.

4.1. A DC/DC konverter felépítése

Ebben a részben a DC/DC konverter általános felépítésével, működésének alapelveivel és a méretezéssel foglalkozunk. A fejezetben megtaláljuk mindazt az elméleti és gyakorlati tudást, mely szükséges az áramkörök megtervezéséhez.

4.1.1. Blokkvázlat, működés

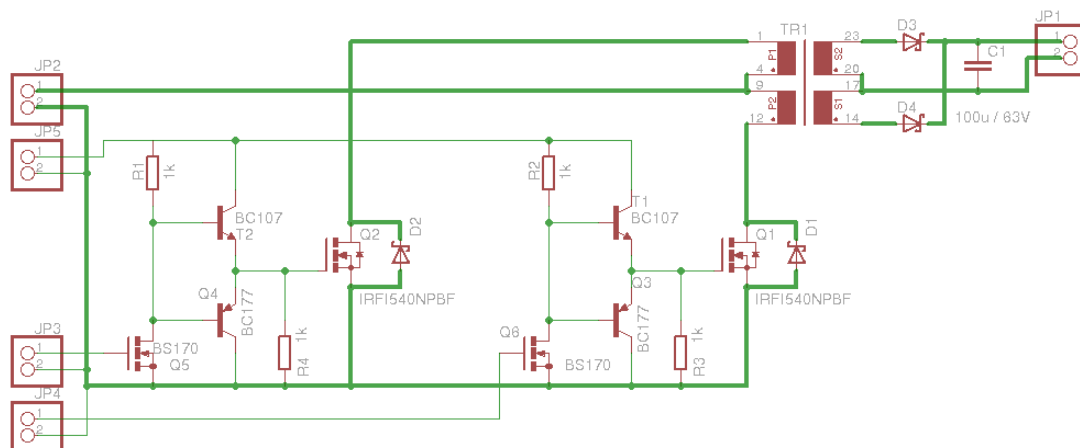
A konverter működését a következő blokkvázlat alapján érthetjük meg a legegyszerűbben:



A bejövő DC feszültséget egy PWM üzemű kapcsolóeszközzel megszaggatjuk, váltakozó feszültségé alakítjuk. Az így kapott jelet a transzformátor segítségével olyan feszültségszintre transzformáljuk, amely alkalmas a szuperkapacitások töltésére. Egyenirányítás és pufferekés után egyenfeszültséghez jutunk. Az ARM processzor feladata, hogy a PWM végrehajtót-beavatkozót úgy vezérelje, hogy a terhelésként kapcsolt szuperkapacitásokat konstans árammal töltsen. Az áram értéke a PWM jel kitöltési tényezőjével szabályozható.

4.1.2. Kapcsolási rajza

A konverter „középpontjában” egy transzformátor található. Mivel ez nem képes egyenáramú jelet transzformálni, sőt, az egyenáram károsan befolyásolja a vasmag működését (könnyen telítésbe viheti), ezért olyan felépítést kell választanunk, mely lehetővé teszi azt, hogy a vasmag hiszterézisgörbéjét mindkét irányban (szimmetrikusan) igénybe tudjuk venni. Egy lehetséges megoldás a következő ábrán látható:

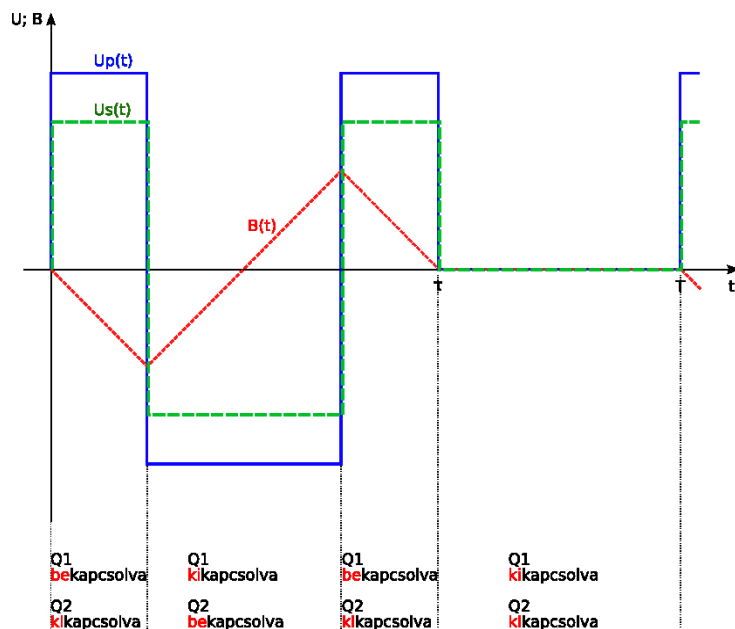


A bal oldali csatlakozón érkező PWM jelek (JP3 és JP4) erősítését a kis teljesítményű FET-ek és bipoláris tranzisztorok végzik. Erre azért van szükség, mert a nagyteljesítményű MOSFET-ek (Q1 és Q2) bemeneti kapacitásait néhány nanoszekundum alatt kell 10-15 V-ra tölteni, vagy kisütni. Hogy ezt megtehesük, nagy áramot kell a GATE kapacitásba pumpálni, illetve onnan kiszívni.

A végtranzisztorok feladata, hogy a JP2-n érkező 270 V-os tápfeszültséget hol az egyik, hol a másik primer tekercsrészre (P1 és P2) kapcsolják. Ha pontosan annyi ideig kapcsolja a Q2-es FET a P1 tekercset tápfeszültségre, mint a Q1-es FET a P2 tekercset, akkor 1 (vagy több) teljes periódusra véve nem fog egyenáram folyni a tekercseken, így megfelelő méretezés esetén nem megy telítésbe a vasmag.

A szekunder oldalon található Schottky-diódák végzik a transzformált feszültség egyenirányítását. A C1 kondenzátor a jel zajtartalmát csökkenti.

A PWM kapcsolójel és a megjelenő feszültségek, indukciók az alábbi ábrán látható:



Ha a Q1-es FET kinyit, áram indul meg a P2-es tekercsen, mely mágneses indukciót okoz a vasmagban. A primer és a szekunder feszültség egyenesen arányos az indukciót változással (a pontos leírással a

4. A DC/DC konverter kialakítása

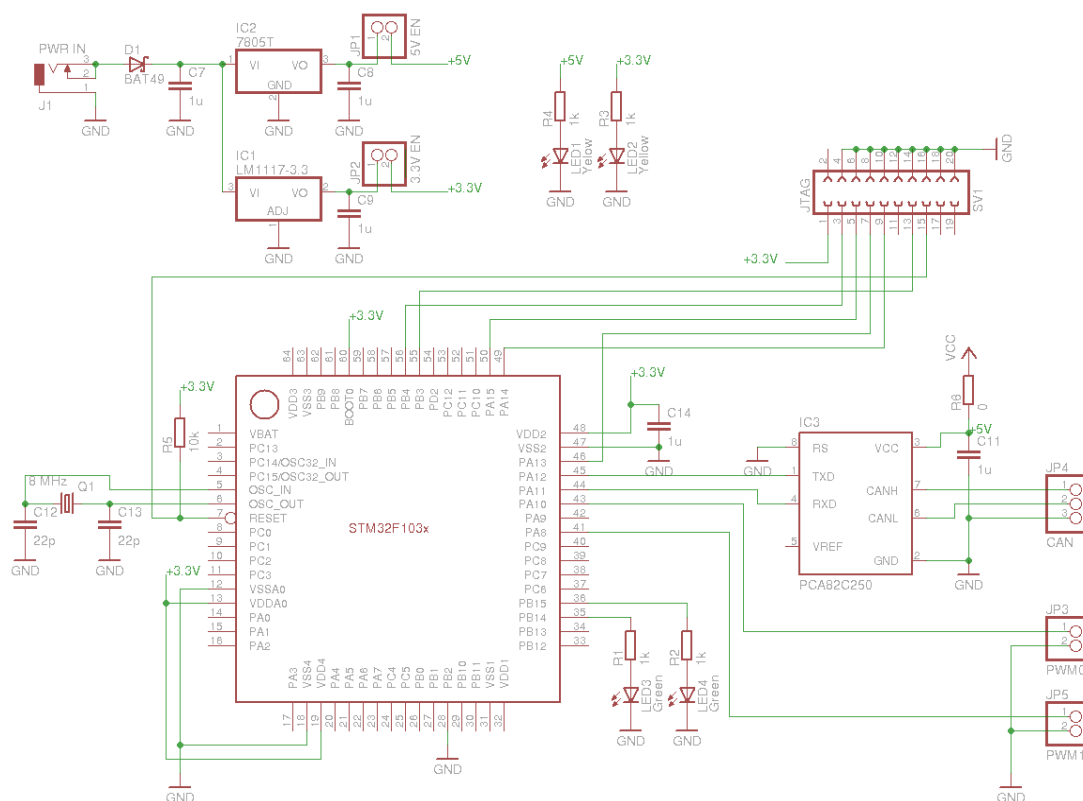
következő részben ismerkedhetünk meg). A menetszámok változtatásával befolyásolhatjuk, hogy mekkora legyen az indukált szekunder feszültség a primer feszültséghez képest.

A szekunder feszültség értékét azzal is befolyásolhatjuk, hogy mekkora a τ értéke a T-hez (periódusidő) képest. A szekunder feszültség és a τ között is egyenes az arányosság. A pontos τ előállítása a PWM generátor feladata (a részletek a 4.2.2. fejezetben találhatók).

A szekunder feszültségre jó közelítéssel igaz a következő összefüggés:

$$U_{szekunder} = U_{primer} \cdot \frac{n_{szekunder}}{n_{primer}} \cdot \frac{\tau}{T}$$

A PWM jelet szolgáltató, ARM Cortex-M3 mikrovezérlő kőre épülő áramkör kapcsolási rajza a következő ábrán látható. A megoldás szinte teljesen megegyezik a soros-CAN átalakító kapcsolási rajzával.



4.1.3. A transzformátor méretezése

Mivel elég nehéz jól használható, de közérthető leírást szerezni a transzformátor méretezéséről, ezért úgy gondolom, hogy kezdjük az egész folyamatot az alapoktól. Jelen esetben ezt az elektromágneses jelenségeket leíró Maxwell-egyenleteket jelenti:

A Maxwell-egyenletek:

$$\text{I. } \oint_{(g)} \vec{H} \cdot d\vec{s} = \int_{(A)} \left(\vec{J} + \frac{d\vec{D}}{dt} \right) \cdot d\vec{A}$$

$$\text{II. } \oint_{(g)} \vec{E} \cdot d\vec{s} = -\frac{d}{dt} \int_{(A)} \vec{B} \cdot d\vec{A}$$

$$\text{III. } \oint_{(A)} \vec{B} \cdot d\vec{A} = 0$$

$$\text{IV. } \oint_{(A)} \vec{D} \cdot d\vec{A} = \int_{(V)} \rho \cdot dV$$

$$\text{V. } \vec{D} = \epsilon \cdot \vec{E}$$

$$\text{VI. } \vec{B} = \mu \cdot \vec{H}$$

$$\text{VII. } \vec{J} = \sigma \cdot \vec{E}$$

$$\text{VIII. } w = \frac{1}{2} \cdot (\vec{B} \cdot \vec{H} + \vec{D} \cdot \vec{E})$$

Ezek közül nekünk a II. egyenlet különösen érdekes, mert ez teremt kapcsolatot az indukció felületi integráljának változása és az indukált elektromos térerősség vonal menti integrálja között.

$$\oint_{(g)} \vec{E} \cdot d\vec{s} = -\frac{d}{dt} \int_{(A)} \vec{B} \cdot d\vec{A}$$

Esetünkben az elektromos térerősség (E) független attól, hogy a vezeték mely szakaszán nézzük, ezért az „E” kivihető az integráljel elé. Ugyanez a mágneses indukcióra is igaz: az is ugyanakkora értékű a vasmag keresztmetszetének minden négyzetmilliméterén.

$$E \cdot \oint_{(g)} ds = -\frac{d}{dt} (B \cdot \int_{(A)} dA)$$

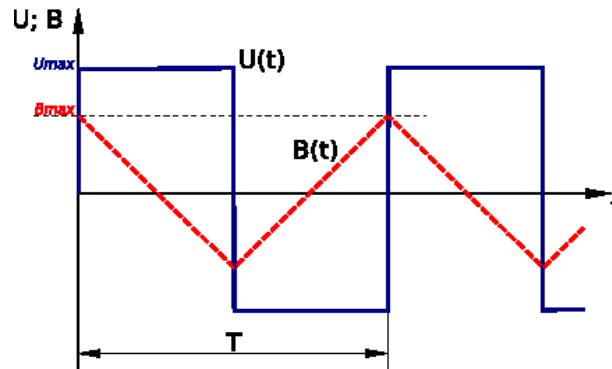
A bal oldalon egy feszültség jellegű mennyiség, az (egy menetre vett) indukált feszültség áll. A jobb oldalon az infinitezimálisan kis felületek összege a teljes felületet adja. A felület nem időfüggő, így írhatjuk a differenciálás elé is.

$$U_1 = -A \cdot \frac{dB}{dt}$$

A teljes, „n” menetes tekercsre vett feszültség így határozható meg:

$$U = -n \cdot A \cdot \frac{dB}{dt}$$

Mivel a vasmagban az indukció lineárisan fog nőni, illetve csökkenni, így a differenciálás valójában osztással egyszerűsödik. Az egyenletes változásnak az az oka, hogy négyszögjellel hajtjuk meg a primer tekercset, vagyis mindkét félperiódusban egy „konstans” feszültség áll a bal oldalon, ami akkor és csak akkor lehetséges, ha a jobb oldalon a $\Delta B/\Delta t$ is konstans („n” és „A” definíció szerint állandó).



$$U = -n \cdot A \cdot \frac{\Delta B}{\Delta t}$$

Ha a vasmagot maximálisan ki akarjuk használni (és miért ne tennénk, hiszen a teljes hiszterézisgörbe ki van fizetve), akkor az indukcióváltozás maximális értéke éppen B_{max} lehet. Az az idő, ami alatt ezt az indukcióértéket el kell érni (Δt_{max}) a periódusidő fele. Ezt a négyszögjel frekvenciájával is kifejezhetjük:

$$\Delta B = B_{max} \quad \Delta t_{max} = \frac{T_{max}}{2} = \frac{1}{2 \cdot f_{min}}$$

Nem jelent problémát, ha nagyobb frekvenciájú négyszögjelet használunk, mint az f_{min} értéke, legfeljebb az indukció nem fogja elérni (vagy túllépni) a B_{max} értéket. Arra azonban vigyázzunk, hogy ne lépjük túl a katalógusban megadott maximális frekvenciát sem.

Az indukált / elvárt maximális primer feszültség a fent elmondottak alapján a következőképpen is felírható:

$$|U| = -n \cdot A \cdot \frac{B_{max}}{\frac{T_{max}}{2}} = 2 \cdot n \cdot A \cdot B_{max} \cdot f_{min}$$

A primer tekercsre kapcsolható négyszögjel amplitúdója:

$$|U_{primer}| = 2 \cdot n_{primer} \cdot A \cdot B_{max} \cdot f_{min}$$

A szekunder tekercsben indukálódó feszültség maximális értéke (maximális primer feszültség esetén).

$$|U_{szekunder}| = 2 \cdot n_{szekunder} \cdot A \cdot B_{max} \cdot f_{min}$$

Könnyen belátható, hogy a két egyenlet hányadosa éppen a 7. osztályban fizikából tanult transzformátor-átvételi tényezőt adja.

Ha az egyenlet bal oldalára a feszültségek és a menetszámok hányadosát rendezzük, akkor az egyenlet jobb oldalán egy olyan kifejezést kapunk, amely az adott vasmagra jellemző paraméter, és adott frekvencián állandó.

$$Y = \frac{|U|}{n} = \frac{|U_{primer}|}{n_{primer}} = \frac{|U_{szekunder}|}{n_{szekunder}} = 2 \cdot A \cdot B_{max} \cdot f_{max}$$

Fizikailag ez a paraméter azt mondja, hogy a vasmag köré tekert egy menetes „tekercsre” mekkora amplitúdójú négyzögjel kapcsolható anélkül, hogy a vasmag telítésbe menne. A jelenleg rendelkezésemre álló vasmag „Y” paramétere a következő:

$$B_{max} = 0,1 \text{ T (katalógusadat)}$$

$$f = 25 \text{ kHz}$$

$$A = 350 \text{ mm}^2 \text{ (katalógusadat)}$$

$$Y = \frac{|U|}{n} = 2 \cdot A \cdot B_{max} \cdot f_{max} = 2 \cdot 350 \cdot 10^{-6} \cdot 0,1 \cdot 25 \cdot 10^3 = 1,75 \frac{\text{V}}{\text{menet}}$$

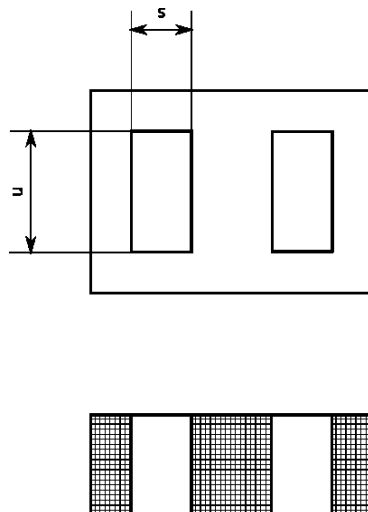
Jelen számítás elvégzésekor maximálisan kihasználtuk a vasmagot, de ez semmiképpen sem célszerű. A továbbiakban $Y = 1,5 \text{ V}/_{\text{menet}}$ értékkel számolok.

Fontos kérdés tudni, hogy a transzformátoron, mint négypóluson mekkora teljesítmény haladhat keresztül. Feltételezzük, hogy a transzformátor ideális négypólus (helyes méretezés esetén) közelebb áll az ideálshoz, mint bármelyik más elektronikai eszközünk: egy jó transzformátor hatásfoka 95 – 99%).

$$P = U \cdot I = Y \cdot n \cdot J \cdot A_{vezeték} = Y \cdot n \cdot J \cdot \frac{A_{tekercs \text{ rézfelülete}}}{n} = Y \cdot J \cdot \frac{A_{ablak}}{2} \cdot k$$

A bemeneten vagy a kimeneten mérhető (látszólagos) teljesítmény a feszültség és az áram effektív értékének szorzata. A feszültség az előbb kiszámított „Y” paraméterrel is felírható. Az áram értékét úgy is meghatározhatjuk, hogy az áramsűrűség értékét megszorozzuk a rézvezeték keresztmetszetével. A rézvezeték keresztmetszet nem más, mint a tekercselés teljes (és hatásos) rézfelületének „n”-ed része. A hatásos rézfelület alatt azt értjük, ahol az áram ténylegesen áthalad. Ez persze kisebb, mint az az „ablakméret”, amit a menetek átdőfnek, ezért egy „k” konstans vezetünk be, ami azt mutatja, hogy a hatásos rézfelület mekkora a transzformátor által mechanikailag meghatározott ablakhoz (az ábrán látható képen az ablak méretei: „s” és „u”, tehát az ablak keresztmetszete: $A_{ablak} = s \cdot u$) viszonyítva. A „k”-t ablakhasználati tényezőnek nevezzük.

Természetesen az ablak két részre oszlik, az egyik része a primer tekercselés, a másik része szekunder tekercselés számára van fenntartva.



Ismét eljutottunk egy olyan képlethez, melyben csak olyan paraméterek szerepelnek, melyek előre megadhatók: az „Y” értékét a vasmag katalógusadataiból számoltuk, az áramsűrűség értékét tapasztalat szerint 5 A/mm²-re célszerű választani, az ablakméret pedig mechanikai adat. Ezzel megadható az a maximális teljesítmény, mely átvihető a transzformátor segítségével.

A saját vasmagomra ez az érték:

$$P = Y \cdot J \cdot A_{ablak} \cdot k = 3 \cdot 5 \cdot 10^6 \cdot \frac{360 \cdot 10^{-6}}{2} \cdot 0,25 = 675 \text{ W}$$

Ez számottevő teljesítmény, de vegyük észre, hogy a például egy 1 kW-os DC/DC konverterhez 2 darab transzformátor szükséges.

Az utolsó dolog, amit feltétlen meg kell említeni a szkin-hatás. A név arra utal, hogy a frekvencia növekedésével a vezetőben folyó áram egyre inkább a vezető felülete („bőre”) felé koncentrálódik. Azt a mélységet, ahol az áram értéke a felületen folyó áram értékének e-ed részére csökken, behatolási mélységnek nevezzük, és így számítjuk:

$$\delta = \sqrt{\frac{2 \cdot \rho}{\omega \cdot \mu}}$$

A képletben a δ a behatolási mélység méterben kifejezve, a ρ a vezető fajlagos vezetőképessége, ω a váltakozó áram körfrekvenciája és μ a vezető mágneses permeabilitása ($\mu_0 \cdot \mu_r$).

* * *

A megépítendő vészhelyzeti energiaellátás áramkörének bemenő feszültsége 270 V \pm 10 %. Mint később látni fogjuk, nem lehet a piacon olyan huzalt beszerezni, amely szükséges lenne a transzformátorok tekercseléséhez, ezért a vészhelyzeti energiaellátás 1:10-es modelljét fogom megépíteni, illetve ha sikerül megfelelő huzalt vásárolnom, akkor egy elem cseréjével könnyen „átállhatok” az 1:1-es áramkörre. Az elméleti számításokat mindkét esetre elvégzem.

Fontos megjegyezni azt, hogy nem szeretném, ha a PWM generátor bármikor is 100 %-os kitöltési tényezővel dolgozna, ezért a maximális kitöltési tényezőt ($D_{\max} = \tau / T$) 80 %-ban határozom meg.

A primer menetszám meghatározásakor arra a feszültségre kell méretezni a transzformátort, amely hatására a vasmag még éppen nem megy telítésbe (30 V illetve 300 V) .

Az áramkörnek még a legalacsonyabb bemenő feszültség (ebben az esetben 24 V illetve 240 V) esetén is megfelelően kell működnie, szolgáltatnia kell a névleges kimenő feszültséget (30 V illetve 300 V). Ezt a szabályt majd a szekunder menetszám meghatározásánál kell betartanunk.

A számítások során $J = 5 \text{ A/mm}^2$ áramsűrűséggel és $k = 0,4$ ablakhasználati tényezővel számolok.

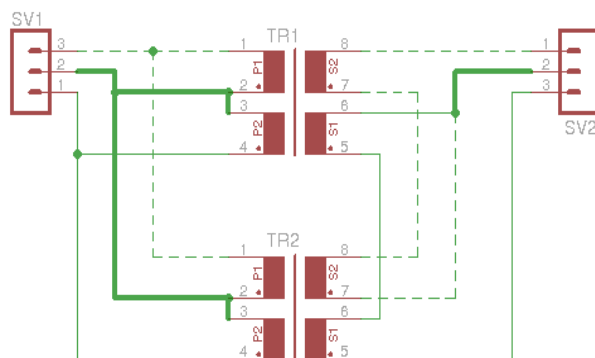
Paraméter	1:10-es modell	Valóságos áramkör
U_{be} : Bemenő feszültség [V_{DC}]	$27 \text{ V} \pm 10\% [24 \dots 30]$	$270 \text{ V} \pm 10\% [240 \dots 300]$
U_{ki} : Névleges kimenő feszültség [V_{DC}]	30	300
I_{ki} : Maximális terhelőáram [A_{DC}]	3	3
Kapacitásban tárolt energia [kWs]	1	10
n_p : Primer menetszám	$n_p = U_{be \max} / Y / D_{\max} =$ $30 / 1,5 / 0,8 =$ 25 menet	$n_p = U_{be \max} / Y / D_{\max} =$ $300 / 1,5 / 0,8 =$ 250 menet
n_{sz} : Szekunder menetszám	$n_{sz} = n_p \cdot U_{ki} / (U_{be \min} \cdot D_{\max}) =$ $25 \cdot 30 / (24 \cdot 0,8) =$ 39 menet	$n_{sz} = n_p \cdot U_{ki} / (U_{be \min} \cdot D_{\max}) =$ $250 \cdot 300 / (240 \cdot 0,8) =$ 390 menet
A_p : Primer tekercs huzalkeresztmetszete [mm^2]	$A_p = I_{ki} \cdot n_{sz} / n_p / J / 2 =$ $3 \cdot 30 / 25 / 5 / 2 =$ 0,36 mm^2	$A_p = I_{ki} \cdot n_{sz} / n_p / J / 2 =$ $3 \cdot 300 / 250 / 5 / 2 =$ 0,36 mm^2
A_{sz} : Szekunder tekercs huzalkeresztmetszete [mm^2]	$A_p = I_{ki} / J / 2 =$ $3 / 5 / 2 =$ 0,3 mm^2	$A_p = I_{ki} / J / 2 =$ $3 / 5 / 2 =$ 0,3 mm^2
d_p : Primer tekercs huzalátmérője [mm]	$d_p = 0,6771 \text{ mm}$	$d_p = 0,6771 \text{ mm}$
d_{sz} : Szekunder tekercs huzalátmérője [mm]	$d_{sz} = 0,6180 \text{ mm}$	$d_{sz} = 0,6180 \text{ mm}$
A_{ablak} : Ablak szükséges keresztmetszete (mm^2)	$A_{ablak} = 2^{**} \cdot 2^{***} \cdot n_p \cdot A_p / k =$ $2 \cdot 2 \cdot 25 \cdot 0,36 / 0,4 =$ 90 mm^2	$A_{ablak} = 2 \cdot 2 \cdot n_p \cdot A_p / k =$ $2 \cdot 2 \cdot 250 \cdot 0,36 / 0,4 =$ 900 mm^2

Látható, hogy a második esetben (valóságos áramkör) egyetlen transzformátoron nem fér el a tekercselés, ezért osztott tekercselést kell alkalmazni. Ez persze nem jelent alapvető változást az architektúrában.

* Mivel az egyik féltekercsben csak az egyik félperiódusban folyik áram.

** A tekercselés a primer és a szekunder tekercset is tartalmazza.

*** A primer tekercs két féltekercsből épül fel.



A szakdolgozatom készítése során nem tudtam 0,30 és 0,36 mm²-es huzalt vásárolni, ezért az UTP kábel ereit használtam. A transzformátor kiválóan működik, de nem tudok 2×250 + 2×390 menetet feltekercselni még 2 csévetestre sem (összesen 2 vasmagom és 4 csévetestem van), ezért döntöttem az 1:10-es modell megépítése mellett.

Később a transzformátorok cseréje könnyen megoldható, nem jelent semmiféle problémát (sőt, módosítást sem kell eszközölni az áramkörön, kivéve persze a bemenő feszültség növelését).

4.1.4. A szuperkapacitás méretezése, áramhatárolás

Az áramkör a szükséges energiát (külső) szuperkapacitásokban tárolja. A szuperkapacitás olyan kondenzátor, melynek kapacitása farad, tíz-farad, száz-farad... nagyságrendben van. Na már ezek az áramköri elemek viszonylag könnyen beszerezhetők. Működésük teljesen megegyezik a „hagyományos” kondenzátorokkal (a szivárgóáramuk jóval nagyobb, így ezek is idővel elvesztik töltésüket – katalógusadat).

A kondenzátorban tárolt energia könnyen kiszámítható:

Ismeretes, hogy a kondenzátorok kapcsain folyó áram értéke:

$$i(t) = C \cdot \frac{dU(t)}{dt}$$

Ez energia definíció szerint a teljesítmény idő szerinti integrálja:

$$E(t) = \int p(t) dt = \int u(t) \cdot i(t) dt = \int u(t) \cdot C \cdot \frac{du(t)}{dt} dt = C \cdot \int u(t) du$$

$$E(t) = \left[\frac{u(t)^2}{2} \right]_0^U = \frac{1}{2} \cdot C \cdot U^2$$

A kondenzátorból kivett energia:

$$E(t) = \left[\frac{u(t)^2}{2} \right]_{U_1}^{U_2} = \frac{1}{2} \cdot C \cdot (U_2^2 - U_1^2)$$

Ha adott idő alatt konstans teljesítményt kívánunk kivenni a kondenzátorból:

$$P \cdot t = \frac{1}{2} \cdot C \cdot (U_2^2 - U_1^2)$$

Ahol U_2 a kezdőfeszültség (amelyre a kondenzátort töltjük), és U_1 a végfeszültség (amelyre a kondenzátor kisülhet anélkül, hogy a kimenő feszültség ± 10 %-os tartományából kilépnénk).

$$10000 \cdot 10 = \frac{1}{2} \cdot C \cdot (300^2 - 240^2)$$

$$100000 = \frac{1}{2} \cdot C \cdot (300^2 - 240^2) = \frac{1}{2} \cdot C \cdot (90000 - 57600) = \frac{1}{2} \cdot C \cdot 32400$$

$$100000 = C \cdot 16200$$

$$C = \frac{100000}{16200} = 6,173 F$$

Ekkora kapacitásérték egyáltalán nem számít különösen nagyknak. A piacon 100 F nagyságrendű kapacitások is elérhetők.

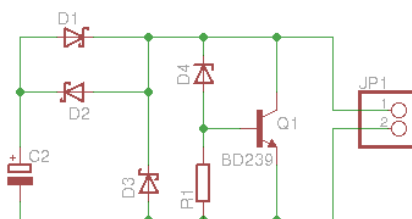
Mivel a szuperkapacitásokra néhány volt vagy néhányszor 10 V kapcsolható, ezért szükséges azok soros kapcsolása. Az eredő kapacitás a részkapacitások értékének és a kondenzátorok számának a hányadosa.

Az általam kiválasztott szuperkapacitásra maximálisan 85 V kapcsolható, így 5 darabra lesz szükségem, egyenként legalább 35 F kapacitással (kapható 100 F-os kivitelben, ezért ezt részesítem előnyben).

Mivel a töltőáramkör kimenetén akár 300-350 V feszültség is megjelenhet, ezért gondoskodni kell a szuperkapacitások egyéni túlfeszültségvédelméről. Elvárások a túlfeszültségvédő áramkörrel kapcsolatban:

- Ne engedje, hogy a kondenzátor a megengedett maximális feszültség fölé töltődjön.
- Ne engedje, hogy a kondenzátor negatív feszültségre töltődjön.
- Az áramkör „kívülről” egy normál kondenzátornak látszon.

Egy lehetséges megvalósítást mutat a következő ábra:



Az áramkör működése rendkívül egyszerű: amíg a kondenzátor kapocsfeszültsége nem éri el a D4 Zener-dióda letörési feszültségét + 0,6 V-ot, addig a Q1 tranzisztor zárva marad, és a D2 Schottky-diódán keresztül töltődik a kondenzátor. Ha a kondenzátor kapocsfeszültsége eléri az előbb említett feszültséget, a Q1 tranzisztor kinyit, és „elszívja” az áramot a C2 kapacitástól, ami nem tud tovább töltődni.

Kisütéskor a Q1 lezárt állapotban marad, és az áram a D1 diódán át elhagyja a szuperkapacitást. A két, antiparallel kapcsolt Schottky-dióda azért szükséges, hogy némi érzéketlenségi tartományt hagyjon a tranzisztor számára.

A kondenzátor negatív feszültségre töltődését a D3 dióda akadályozza meg.

4.2. Szoftver keretrendszer

Ebben a részben arról lesz szó, hogy hogyan valósítottam meg a vészhelyzeti áramforrás töltőáramkörének működtető szoftverét. Először az analóg-digitális átalakító áramkörök működését ismerhetjük meg, majd megnézzük, hogy hogyan tudjuk a transzformátor működéséhez szükséges PWM jelet előállítani.

A fejezet hátralevő részében kiválasztjuk a megfelelő szabályozó algoritmust, és összeállítjuk az előbb említett részekből a főprogramot és a CAN vezérlő megszakításkiszolgáló rutinját.

4.2.1. Az A/D átalakító használata

Az analóg-digitális átalakító használata nem bonyolultabb, mint a többi STM32 eszköze. A gyártó cég által firmware library számos, előre megírt függvénnyel támogatja munkánkat. Az A/D átalakító használatához legalább két függvényt kell készítenünk: az egyik inicializálja az eszközt, a másik pedig (adott időpillanatban) elvégzi az átalakítást, és visszaadja az A/D átalakító bemenetén mérhető feszültség számmá konvertált értékét.

Tudnunk kell, hogy az STM32 számos analóg bemenettel rendelkezik. Ezek közül egy multiplexerrel választjuk ki, hogy melyiket szeretnénk használni.

Másrészt a mikrovezérlő lehetővé teszi, hogy ne csak egy csatornával dolgozzunk, hanem akár nyolccal. Ekkor az átalakítást úgy végzi el a mikrovezérlő, hogy egyenként kiválasztja az aktuális csatornát, és elvégzi az A/D konverziót. Inicializáláskor úgy állítjuk be a paramétereket, hogy csak a szükséges számú csatornán végezze el a mikrovezérlő a konverziót.

Az A/D átalakító használatát segítő forráskód a következő (rendre: `adc.h` és `adc.c`):

```
// -----  
// This header file contains the function prototypes for the Analog-Digital  
// Converter.  
// -----  
  
#ifndef __ADC_H__  
#define __ADC_H__  
  
// -----  
// This function initializes the A/D converter.  
// -----  
  
void adc_init();  
  
// -----  
// Starts A/D conversion.  
// -----  
  
unsigned int adc_convert();  
  
#endif
```

Az előbb deklarált függvények definíciója:

```
// -----  
// This header file contains the function prototypes for the Analog-Digital  
// Converter.  
// -----
```

```

#include <adc.h>
#include <stm32f10x_adc.h>
#include <stm32f10x_gpio.h>
#include <stm32f10x_rcc.h>

// -----
// This function initializes the A/D converter.
// -----

void adc_init() {
    GPIO_InitTypeDef GPIO_InitStructure;
    ADC_InitTypeDef ADC_InitStructure;

    /* Enable ADC1, ADC2 and GPIOC clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_ADC2, ENABLE);
    /* ADCCLK = PCLK2/8 = 12 MHz */
    RCC_ADCCLKConfig(RCC_PCLK2_Div8);

    /* Configure PC.00 and PC.01 (ADC Channel10 and 11) as analog input */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    /* ADC1 configuration -----*/
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfChannel = 1;
    ADC_Init(ADC1, &ADC_InitStructure);

    /* ADC1 regular channels configuration */
    ADC_RegularChannelConfig(ADC1, ADC_Channel_10, 1, ADC_SampleTime_1Cycles5);

    /* Enable ADC1 */
    ADC_Cmd(ADC1, ENABLE);

    /* Enable ADC1 reset calibration register */
    ADC_ResetCalibration(ADC1);

    /* Check the end of ADC1 reset calibration register */
    while(ADC_GetResetCalibrationStatus(ADC1));

    /* Start ADC1 calibration */
    ADC_StartCalibration(ADC1);

    /* Check the end of ADC1 calibration */
    while(ADC_GetCalibrationStatus(ADC1));
}

// -----
// Starts A/D conversion.
// -----

unsigned int adc_convert() {
    /* Start ADC1 Software Conversion */
    ADC_Cmd(ADC1, ENABLE);

    /* Check the end of ADC1 calibration */
    while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == 0);
}

```

```

ADC_ClearFlag(ADC1, ADC_FLAG_EOC);

return ADC_GetConversionValue(ADC1) & 4095;
}

```

Az alapbeállítást végző (`adc_init`) függvény először órajellel látja el a perifériát (a PLL frekvenciáját egy 8-as előosztón keresztül juttatja az A/D átalakító órajelbemenetére, így $12 \text{ MHz} / 14^* = 857,15 \text{ ks/sec}$ sebességű mintavételt tesz lehetővé), majd a megfelelő kimeneteket úgy konfigurálja, hogy azok képesek legyenek analóg jelek fogadására. Ezután beállítja, hogy az átalakító csak a megfelelő számú csatornáról vegyen mintát, de azt is csak szoftver trigger hatására tegye.

Ezt követően beállítja a szükséges bemeneteket. Végezetül engedélyezi és kalibrálja az A/D átalakítót.

Az `adc_convert` függvény szolgáltatja az előbb említett szoftver trigger, amely elindítja az átalakítást. A függvény visszatérési értéke az aktuális csatorna bemenetén mérhető analóg érték számmal kifejezett értéke.

Az `adc_convert` függvényt a főprogram periodikusan meghívja, hogy így frissítse az bementek értékét tároló változókat. A szabályozó algoritmus ezen változókat használja a szabályozási paraméterek megállapításához.

4.2.2. A PWM jel előállítás

A vészhelyzeti áramforrás központi eleme (az ARM alapú mikrovezérlő mellett) a teljesítménytranszformátor. Azért, hogy teljesen ki tudjuk használni a B-H karakterisztikáját, speciális PWM jelre van szükségünk (ezt már láttuk a 4.1.2. fejezetben). Mivel a hardveres PWM generátorok nem tudnak olyan jelez szolgáltatni, mint amilyenre szükségünk lenne, nekünk magunknak kell azt előállítani. Ez szerencsére nem okoz komoly problémát, mert a processzor „magjába” integrált SYSTICK számlálóval könnyen kiválthatjuk a megszakítást másodpercenként 25.000-szer.

A megszakítási rutinnak nincs más dolga, mint beállítani a kimenetet, várni egy meghatározott ideig, aztán módosítani a kimenetet, megint várni... és a végén újraszámolni a várakozási értékeket (ez már a szabályozó algoritmus része).

A SYSTICK számláló használatával már foglalkoztunk a 2.4. fejezetben, így most erre már nem térnék ki.

Ahhoz, hogy a megszakítási rutin másodpercenként 25.000-szer fusson le, a SYSTICK számlálót megfelelőképpen kell beállítani. Erre a `systick_init` függvény szolgál, mellyel már előzőleg megismerkedtünk. A `systick_init` függvényt a `sysinit` hívja meg a mikrovezérlő kódjának indulásakor 25000-es paraméterrel.

A kiszolgáló rutin a következőképpen néz ki (`irq.c`):

```

// -----
// This file contains the Interrupt Service Routines (ISR)
// used by the firmware.
// -----

#include <config.h>
#include <gpio.h>

// -----
// PWM outputs
// -----

```

* A konverzió 14 órajel ideig tart.


```

#define PWM_OUT0 (1 << 8)
#define PWM_OUT1 (1 << 10)

extern volatile unsigned int pwm_duty_cycle;

extern volatile unsigned int current_value;
extern volatile unsigned int current_ref_value;

extern volatile unsigned int output_value;
extern volatile unsigned int output_ref_value;

volatile unsigned int counter = 0;

// -----
// Does some delay according to pwm_duty_cycle.
// -----

void inline delay() {
    unsigned int counter;

    for (counter = 0; counter < pwm_duty_cycle; counter++) {
        asm("nop");
    }
}

// -----
// ISR of the SYSTICK timer (generated the appropriate PWM signals).
// -----

void systick() {
    gpioa_clear(PWM_OUT0); delay(); gpioa_set(PWM_OUT0);
    gpioa_clear(PWM_OUT1); delay(); gpioa_clear(PWM_OUT1);
    gpioa_clear(PWM_OUT1); delay(); gpioa_set(PWM_OUT1);
    gpioa_clear(PWM_OUT0); delay(); gpioa_set(PWM_OUT0);

    // ITT TALÁLHATÓ A SZABÁLYOZÓ ALGORITMUS KÓDJA, LÁSD KÉSŐBB.
}

```

A fent látható `systick` függvény nem csinál mást, mint kinyitva a Q_1 FET-et, majd várakozik egy meghatározott ideig: pontosan addig, amíg végrehajt `pwm_duty_cycle` számú NOP³⁹ utasítást. Később a `pwm_duty_cycle` változtatásával tudjuk a várakozás mértékét befolyásolni.

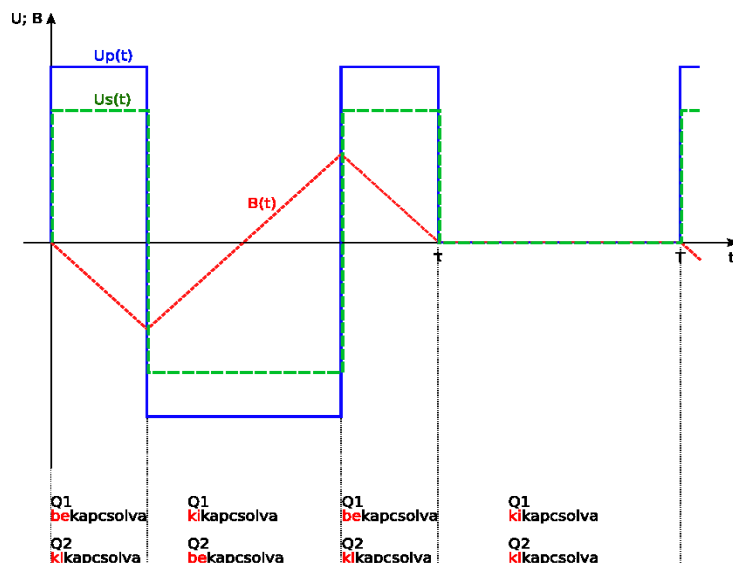
Aztán bezárja a Q_1 -et. Ezalatt a vasmagban a mágneses indukció egyenletesen növekedve elérte B_{\max} értékét.

Kinyitja a Q_2 FET-et, majd kétszer annyi ideig vár, mint az előbb. Ennek hatására az indukció $+B_{\max}$ -ról $-B_{\max}$ -ra változik. Természetesen Q_2 -t le kell zárnia.

Végül újra kinyitja Q_1 -et, és vár egyszeres ideig. Az indukció $-B_{\max}$ -ról 0-ra növekedik. A Q_1 lezárása nem maradhat el. Könnyen belátható, hogy a várakozás értékével a PWM jel kitöltési tényezőjét változtathatjuk, ezzel pedig a kimenő jel effektív értékét szabályozhatjuk.

Emlékeztetőül nézzük meg újra azt az ábrát, amely a feszültség és az indukció változását mutatja az idő függvényében:

39 NOP: No Operation: üres művelet, dolgozik a processzor, de „semmi látszatja”, azaz nem történik érdemi műveletvégzés.



4.2.3. A szabályozó algoritmus kiválasztása

A megszakítási rutin másik feladata, hogy a terhelőáram aktuális értékének megfelelően csökkentse vagy növelje a PWM kitöltési tényezőt (a várakozási időket: a `pwm_duty_cycle` értékét). Ehhez az A/D átalakító segítségével meg kell mérnie az áramsöntön eső feszültséget, valamint a szuperkapacitás kapcsai közötti potenciálkülönbséget, és a szabályozó algoritmusnak megfelelően dönteni kell a várakozási idők módosításáról.

Az előző fejezetben mutatott megszakítási rutin a következő néhány sorral egészül ki:

```
if ((counter % 1024) == 0) {
    if ((output_value - output_ref_value) < 0) &&
        ((current_value - current_ref_value) < 0)) {
        pwm_duty_cycle += STEP_VALUE;
    } else {
        pwm_duty_cycle -= STEP_VALUE;
    }
}

counter++;
```

Az algoritmus egy nagyon egyszerű, mégis hatékony megoldást mutat: a kimenő áram értékétől⁴⁰ és a szuperkapacitás töltöttségének⁴¹ megfelelően egy előre meghatározott értékkel (`STEP_VALUE`) módosítja a `pwm_duty_cycle` változó értékét, vagyis megváltoztatja a kimenő négyesjegyű kitöltési tényezőjét.

Nem lenne célszerű az ellenőrzést-szabályozást másodpercenként 25.000-szer elvégezni, hiszen 40 μ s alatt a szabályozási kör többi eleme nem képes reagálni a változásra. Annak érdekében, hogy lassítsuk a szabályozó működését, egy számlálót növelünk (`counter`), és amikor ennek értéke 1024-gyel osztva nullát ad maradékul (másodpercenként ez nagyjából 25-ször fordul elő), elvégezzük az ellenőrzést és a módosítást.

Könnyen belátható, hogy az algoritmus egy integráló típusú szabályozást valósít meg. Kétségtelen előnye, hogy a maradék szabályozási eltérés értéke egy idő után nullára csökken, hátránya, hogy konstans hiba esetén beavatkozó jel a végtelenségig növekedhet vagy csökkenhet. Ennek megakadályozására (és a maximális

⁴⁰ Ezt igyekszik állandó értéken tartani, amíg a szuperkapacitás fel nem töltődik.

⁴¹ Az algoritmus nem engedi, hogy a maximálisan megengedett feszültség fölé töltődjön a kapacitás.

értékek betartásának céljából) határolni kell a beavatkozó jelet: ezt az `if` utasítás két ága valósítja meg. Az „igaz” ág szabályoz, a „hamis” ág határol.

Így tulajdonképpen egy feszültséghatárolt áramgenerátorhoz, vagy áramhatárolt feszültséggenerátorhoz jutunk. A névleges értékeket a `output_ref_value` és a `current_ref_value` globális változó tárolja, ezek értéke CAN buszon keresztül lekérdezhető és beállítható.

Az integrálási idő tényleges értékét úgy növelhetjük, hogy csökkentjük a hiba „modulusát”, vagyis azt az értéket, amellyel a hiba értékét elosztjuk. Ha így teszünk, akkor „nagyobb lépésekkel” haladunk a névleges érték felé, ekkor azonban a szabályozási körünk instabillá válhat. A szabályozási paraméterek megállapítása egy külön könyv témája lehetne.

4.2.4. A főprogram működése

A főprogram a megszakítási rutintól függetlenül működik. Három fontos feladatot lát el:

- Periodikusan elvégzi az A/D átalakítást (azért nem a megszakítási rutin végzi azt, mert így a két folyamat párhuzamosítható, így csökken a teljes rendszer erőforrásigénye).
- Szükség esetén, ha a tápenergia ellátás kimarad, akkor a szuperkapacitásokat a terhelésre kapcsolja, így biztosítja, hogy a terhelés ne maradjon energiaellátás nélkül.
- Ellenőrzi, hogy érkezett-e kérés a CAN buszon keresztül (ehhez felhasználja a CAN megszakítást és a már ismertetett queue-kat), ha érkezett, akkor arra megfelelőképpen reagál: módosítja a szabályozási paramétereket, vagy visszaadja a kért változó értékét.

A főprogram feladata a kommunikáció biztosítása a mikrovezérlő és az azt irányító eszköz között. A protokoll működését és a megfelelő forráskódot a következő fejezet tartalmazza.

4.3. Kommunikáció megvalósítása

Ebben a részben azt vizsgáljuk meg, hogy hogyan lehet megfelelően biztonságos, megbízható és gyors kommunikációt megvalósítani a vészhelyzeti áramforrás és a vezérlő berendezés (PC, egyéb mikrovezérlő) között.

A kommunikáció alapvető fontosságú, ugyanis a rosszul megtervezett adatátvitel nem csak megbízhatatlan összeköttetést okoz, hanem esetleg hibás parancs érkezhet az áramforrásnak. Hogy ezt elkerüljem, a CAN választottam fizikai és adatkapcsolati rétegnek. A többi réteg (hálózati, adatkapcsolati, alkalmazási) leírását pedig a következő alfejezetekben találjuk.

4.3.1. Fizikai réteg

A CAN busz fizikai rétegének működését a 3.1.3. fejezetben már megismerhettük. Ennek feladata – definíció szerint –, hogy lehetővé tegye bitek átvitelét a kommunikációban részt vevő eszközök között. Eredetileg úgy tanultuk, hogy "bitek hibamentes átvitelét" kell lehetővé tennie, de a kommunikáció során sosem lehetünk biztosak abban, hogy valóban azt veszi az ellenállomás, mint amit küldünk.

A CAN fizikai rétege egy szimmetrikus érpárból álló buszból, és azt az meghajtó nyitott kollektoros adó-vevő áramkörről áll. A keretek átvitelét a következő részben ismertetendő adatkapcsolati réteg végzi.

4.3.2. Adatkapcsolati réteg

A CAN busz esetében az adatkapcsolati réteg feladata, hogy lehetővé tegye adatblokkok (keretek) átvitelét a kommunikációban részt vevő állomások között.

A CAN vezérlő ettől többet nyújt a felhasználó számára, mert nem csak a keretezést végzi el (keret kezdetének megállapítása, ütközés elkerülése és annak érzékelése, stb.), hanem lehetővé teszi az üzenetek prioritizálását (a fontosabb üzenetek elsőbbséget élveznek a kevésbé fontosakkal szemben), továbbá lehetőséget ad szűrésre, és integritás-ellenőrzésre (CRC) is.

A CAN busz adatkapcsolati részének működésével már részletesen foglalkoztunk a 3.1.3. fejezetben, így ezt most nem tárgyalom részletesen.

Hálózati szempontból annyit kell megemlíteni, hogy az adatkapcsolati réteg csak az arbitrációs rész felső 3 bitjét használja azért, hogy a kereteket prioritási osztályokba sorolja. Ezzel a megoldással 8 prioritási osztály jön létre, ezek közül a 0. a legmagasabb (vészhelyzet) és a 7. a legalacsonyabb (informálás, nyomkövetés). Az arbitrációs mező többi bitjét a felsőbb rétegek (hálózati és szállítási) fogják használni.

4.3.3. Hálózati réteg

Ennek a rétegnek az a feladata, hogy lehetővé tegye az információcserét két állomás között akkor is, ha a két állomás nincsen ugyanabban a fizikai hálózatban. A hálózati réteg "megtalálja" a célállomást, és továbbítja neki a forrás eszköztől származó csomagot (szükség esetén egy közvetítő médiumon, „külső” hálózaton keresztül).

Mivel ebben a dolgozatban nem kívánok különösebben komplex hálózatot kialakítani, azért élek azzal az egyszerűsítéssel, hogy a hálózati réteg az adatkapcsolati réteggel szorosan együttműködve fejt ki a tevékenységét, feladata arra szűkül, hogy két, azonos hálózatban levő eszköz között tegye lehetővé a kommunikációt.

Az eszközök 8 bites címmel rendelkeznek, így 256 hálózati elem megcímzésére van lehetőség. Ez nem jelenti azt, hogy 256 eszközt kellene vagy lehetne egy hálózatba kötni, mindössze szeretném biztosítani a lehetőséget arra, hogy egy eszköznek több címe is lehessen (virtuális eszközök).

A hálózati réteg által megkövetelt fejléc az elmondottakkal összhangban a következőképpen néz ki:

- 3 bit: prioritási osztály (adatkapcsolati réteg)
- **8 bit:** a forrás állomás (virtuális) címe
- **8 bit:** a cél állomás (virtuális) címe

Általában nagyon ajánlott valamilyen hibaellenőrző vagy hibajavító kódot használni az átvitel során, de a CAN interfész ezt hardveres úton megoldja helyettünk (a CRC⁴² ellenőrzést automatikusan elvégzi), így ettől most eltekintek.

Mivel a mikrovezérlőnek nem kell útvonalválasztással (routing) foglalkoznia, ezért a hálózati réteg forráskódja meglehetősen egyszerű lesz.

4.3.4. Szállítási réteg

Mind közül ez a legösszetettebb, hiszen ennek a rétegnek az az elsődleges feladata, hogy az alkalmazástól érkező, tetszőleges hosszúságú üzeneteket a forrás állomás oldalán kis (néhány bájt ... néhány száz bájt) részekre darabolja, ezen részeket a hálózati réteggel együttműködve eljuttassa a célállomáshoz, és ott újra hibamentesen összeállítsa az eredeti üzenetet. Természetesen szükség van a hibás vagy a meg nem érkezett "darabok" újrakérésére, az átvitel vezérlésére, stb.

Mivel a CAN maximálisan 8 adatbájttal rendelkező kereteket tud küldeni, így automatikusan adódik, hogy maximálisan mekkora darabokban tudjuk az alkalmazás üzeneteit a vevő oldalára eljuttatni.

42 CRC: Cyclic Redundancy Check: ciklikus redundanciaellenőrzés

A szállítási réteg kódja az alkalmazás üzenetét blokkokra bontja, és minden ilyen blokkot elküld úgy, hogy a keret fejléce tartalmazza a blokk sorszámának legalsó bitjét, és a blokk hosszát (a 2 egész kitevőjű hatványa). A vevő oldalnak minden keretet nyugtáznia kell, mégpedig úgy, hogy a nyugta fejléce tartalmazza a blokk sorszámának legalsó bitjét. Így az adó és a vevő szinkronizálja magát az adatblokkok és a nyugták segítségével, tulajdonképpen egyfajta kézfogásos üzemmódot valósítanak meg.

Ha egy adatblokk nem érkezik meg, vagy megsérült, akkor a vevő jelzi (nyugtázza), hogy az előző adatblokkok még megjött (mert még az előző „legalsó bitet” küldi el). Ebből az adó tudni fogja, hogy a legutóbbi blokk elveszett, és a legutolsó blokkot újra el kell küldenie.

A teljes CAN-es kommunikációhoz szükséges fejlécformátum a következő:

- 3 bit: prioritási osztály (adatkapcsolati réteg)
- 8 bit: a forrás állomás (virtuális) címe (hálózati réteg)
- 8 bit: a cél állomás (virtuális) címe (hálózati réteg)
- **1 bit:**
 - ha 0: adatokat tartalmazó keretről van szó
 - ha 1: vezérlő információt küldünk
- **1 bit:**
 - A keret sorszámának legalacsonyabb helyiértékű bitje
- **6 bit:** blokkméter (N)
- **2N × 8 bit (CAN esetén 64 bit):** a adatmező (lásd még: alkalmazási réteg)

A vezérlő információkat arra használjuk, hogy jelezzük egy felhasználói üzenet végét, nyugtát küldjünk, vagy módosítsuk a kommunikáció paramétereit. A lehetséges vezérlő információk a következők:

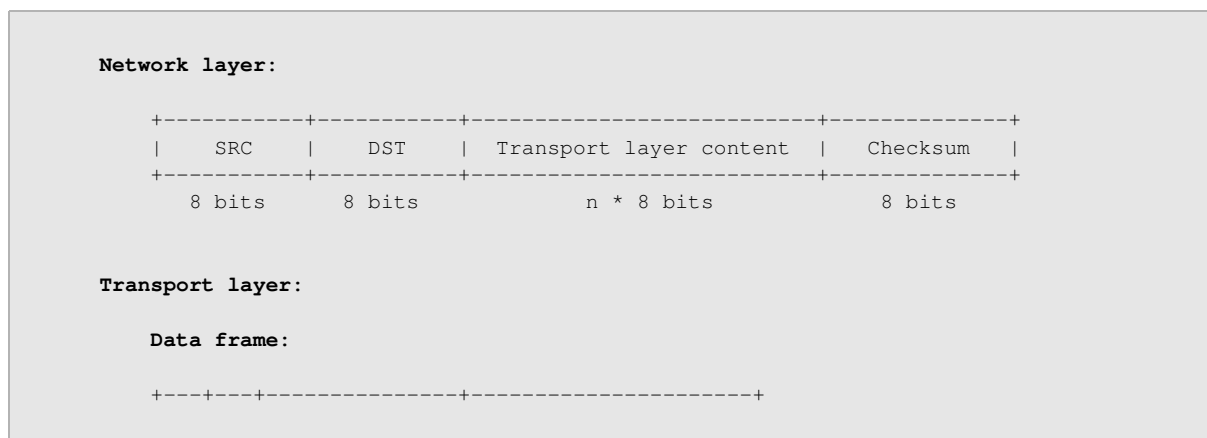
Ha az adó küldi:

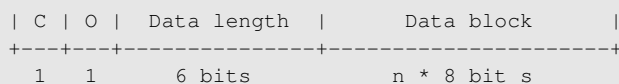
- 0xFF: alkalmazás üzenetének vége, nyugta kérése

Ha a vevő küldi:

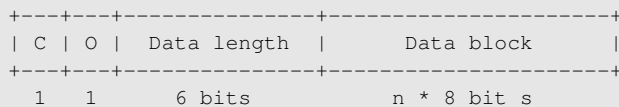
- 0x00: pozitív nyugta
- 0x01: negatív nyugta (hiba történt)

A protokoll fejlécei grafikusán a következő ábrán láthatók:





Control frame:



A szállítási rétegben gyakran hitelesítést és titkosítást is megvalósítanak. Azért szoktak egy viszonylag magas szintű réteget választani, mert a szállítási réteg már garantálja az üzenetek sorrendhelyes és hibamentes átvitelét. Az alkalmazási rétegtől érkező üzenet bármekkora lehet, így célszerű azt szimmetrikus algoritmussal titkosítani (pl. a XOR folyam titkosítóval – stream cipher).

Nyílt kulcsó algoritmussal hitelesítést (RSA-val digitális aláírást) és kulcscserét (Diffie-Hellmannal) is megvalósíthatunk, hiszen lehetőségünk van akár több ezer bites kulcsok, és részeredmények küldésére-fogadására is.

4.3.5. Alkalmazási réteg

A legfelső szintű réteg feladata az absztrakt adatstruktúrák kezelése. A főprogram tartalmazni fogja mindazt a funkcionalitást, amire szükségünk lesz ahhoz, hogy a vészhelyzeti energiaellátás magas szinten kommunikálni tudjon a külvilággal, a többi eszközzel.

4.3.6. A kommunikációs protokoll megvalósítása

A 8 bájt adat, amely a CAN üzenet adatmezőjében foglal helyet, a következő jelentést hordozza:

- 2 bájt: parancs (funkció kód)
 - 0x0000: üres parancs
 - 0x0001: regiszter tartalmának lekérdezése
 - 0x0002: regiszter tartalmának beállítása
 - 0x0003: regiszter tartalmának beállítása és értékének visszaküldése
- 2 bájt: regiszter sorszáma
- 4 bájt: regiszter új értéke (csak beállításkor, egyébként nulla)

Ez azt jelenti, hogy az adatkommunikáció 8 bájtos üzenetek segítségével történik, vagyis az adatkapcsolati réteg fejlécében az adathossz mező 3 lesz. Mivel a felhasználói üzenet mérete teljesen megegyezik az adatkapcsolati réteg által átvihető információmennyiséggel, ezért minden felhasználói üzenet egy darab CAN üzenetté alakul. Tehát a felhasználói üzenet végét minden CAN üzenet végén jelezni kell vezérlő információk segítségével, de ez nem jelent jelentős terhelést a CAN busz számára. A vevő oldalnak minden CAN üzenetet nyugtáznia kell.

A kommunikációhoz szükséges függvények, eljárások forráskódját terjedelmi okokból nem áll módomban közölni, de a CD-melléklet *Firmware* könyvtárában az Olvasó megtalál minden szükséges fájlt.

5. Összefoglalás, végkövetkeztetés

Ebben a dolgozatban azt írtam le, hogy hogyan lehet egy vészhelyzeti áramforrást és az azt kiszolgáló áramköröket-szoftvereket megtervezni, majd megvalósítani.

A dolgozat első részében megfogalmaztam az elvárásokat (specifikáció), lefektettem az elvi alapokat, legalább blokkvázlat szinten, majd lépésről lépésre áttekintettem a rendszer építőelemeit:

A következő fejezetben megvizsgáltam, hogy hogyan alakult ki az áramköröm magvát képző ARM Cortex-M3 mikrovezérlő, hogyan lehet programozni, és hogyan lehet a kódot hibamentesíteni.

A soros-CAN átalakító alapvető fontosságú, mert ezen az áramkörön keresztül tudja a PC tartani a kapcsolatot az áramforrással. Sorra vettem az átalakító megépítésének lépéseit, majd bemutattam néhány képernyőképen keresztül a működését.

A soros-CAN átalakítót leíró fejezetet a vészhelyzeti áramforrás megtervezése követte. Részletesen, az alapoktól kezdve kifejtettem az áramkör, de legfőképp a transzformátor méretezésének módját, majd megadtam a szuperkapacitás méretezéséhez szükséges képleteket és paramétereket.

De semmit sem ér az az áramkör, amely képtelen a külvilággal kapcsolatot tartani. Ezért a legutolsó alfejezetet arra szántam, hogy egy megbízható, biztonságos, de gyors kommunikációs protokollt fejlesszek.

Bátran állíthatom, hogy az elkészült áramkör a specifikációban megfogalmazott igényeket maradéktalanul kielégíti, sőt, bőven túl is teljesíti azokat.

6. Felhasznált szoftverek

- OpenOffice.org 2.4.1 – szövegszerkesztő, szövegformázó
- GCC 4.2.3 (Sourcery G++ Lite 2008q1-126) – GNU Compiler Collection (GNU fordítógyűjtemény)
- Dia 0.96.1 – diagram szerkesztő
- Inkscape 0.46 – vektorgrafikus szerkesztő
- OpenOCD – Open On-Chip Debugger (nyílt lapkán belüli nyomkövető)
- STM32 Firmware Library – STM32 beágyazott programkönyvtár
- Eagle 5.3.0 Light Edition – nyomtatott áramkör tervező

7. Felhasznált irodalom

- Cortex™-M3 Technical Reference Manual – műszaki referencia kézikönyv
- RM0008 STM32 Reference Manual – STM32 referencia kézikönyv
- <http://wikipedia.org/> – A szabad enciklopédia (az ARM processzorok története)
- <http://www.softing.com>⁴³ – a CAN busz képeinek forrása

8. Készítendő képek – Zita

- Serial-CAN eszköz
- CAN üzenet szkópon
- PWM jelek: uC, FET-ek G-je, D-je, trafón és a szekunder oldalon AC, DC
- PWM áramkör
- PWM generátor terheléssel
- Trafók?
- Szabályozás
-

43 <http://www.softing.com/home/en/industrial-automation/products/can-bus/>

9. Frissítési eljárás

- Release szám növelése
- Helyesírás-ellenőrzés
- Elválasztás
- Tartalomjegyzék
- Tárgymutató
- PDF generálás