

CAN buszra fűzhető teleport és időgép megvalósítása
ARM Cortex-M3 alapú mikrovezérlővel

DRAFT

Köszönetnyilvánítás

Ezúton szeretném kifejezni köszönetemet konzulensemnek, __FIXME__-nek (AKI KITALÁLJA, HOGY MIT CSINÁLJAK, AZ LESZ A KONZULENSEM – KOMOLYAN!), és **Krüpl Zsolt** okleveles villamosmérnöknek, akik áldozatos munkájukkal, ötleteikkel és nélkülözhetetlen szakmai tanácsaikkal segítettek a dolgozat megírásában.

Szintén hálával tartozom a lektoromnak, **Körmendi Zita** kommunikációs szakembernek, aki erején felül teljesítve azon fáradozott, hogy a dolgozat mindenfajta – helyesírási, stilisztikai és logikai – hibától mentesen kerülhessen az Olvasó elé.

Köszönet illeti még egyik fő támogatómat, az ST Microelectronics Company-t: munkatársai számos mikrovezérlővel járultak hozzá a diplomamunkám létrejöttéhez.

Tartalomjegyzék

Köszönetnyilvánítás	3
Bevezetés	7
1. A teleport és időgép felépítése	9
2. Az ARM processzorok	10
2.1. Történeti áttekintés	11
2.2. Szoftverfejlesztés ARM processzorra	14
2.2.1. C/C++ fordító fordítása ARM architektúrára	14
2.2.2. C/C++ program írása ARM architektúrára	16
2.2.3. Kód letöltése, hibamentesítés	22
2.3. Gyártóspecifikus hardver kezelése	30
3. USB-CAN átalakító	31
3.1. A CAN busz működése és használata	31
3.2. Kommunikáció a PC-vel: az USB	31
3.3. Az USB-CAN átalakító megépítése	31
4. A teleport és időgép kialakítása	32
4.1. Motiváció, hardver specifikáció	32
4.2. A teleport és időgép felépítése	32
4.2.1. Blokkvázlata, működése	32
4.2.2. Kapcsolási rajza, NYÁK-terve	32
4.2.3. Szoftver keretrendszer	32
4.3. Kommunikáció az átalakítóval és más CAN-es eszközzel	32
5. Továbbfejlesztési lehetőségek	33
6. Összefoglalás, végkövetkeztetés	35
7. Címsor 1	37
7.1. Címsor 2	37
7.1.1. Címsor 3	37
7.1.1.1. Címsor 4	37
7.1.2. Tisztelet Sákjamuni buddhának	37
8. Felhasznált szoftverek	39
9. Felhasznált irodalom	39

Bevezetés

Azért, mert diplomamunka. Leírni, hogy mi lesz, miért és hogyan.

Melléktermékek keletkeznek: egy USB-CAN átalakító és egy szoftver framework.

A „Lorem Ipsum” egy nyomdatechnikai szöveg, melynek csak kitöltő szerepe van, nem része a diplomamunkának.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce purus libero, iaculis sit amet, suscipit id, sagittis in, dolor. Morbi consectetur rhoncus felis. Fusce dolor metus, luctus congue, molestie in, dapibus ut, eros. Suspendisse quam. Mauris id urna a justo fringilla iaculis. In hac habitasse platea dictumst. Aenean pharetra ultrices eros. Etiam rutrum semper nunc. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Aliquam tortor tellus, sollicitudin a, eleifend a, rhoncus in, sem. Pellentesque lobortis felis vitae massa. Phasellus iaculis augue a dui elementum elementum. Mauris risus. Sed sem enim, aliquam accumsan, pharetra vitae, cursus sit amet, erat.

Praesent pulvinar, magna ut ultrices consectetur, nisl dolor imperdiet risus, id pretium purus sapien sed diam. Maecenas ut enim at tortor porta sodales. Nulla eget magna eget tellus venenatis rhoncus. Vestibulum vestibulum. Aliquam semper augue id nulla. Proin accumsan laoreet sapien. Vivamus tristique nisi sed quam. Aliquam eget sapien. Etiam pharetra, nisl id imperdiet adipiscing, lorem elit fringilla lorem, eget pretium augue lacus eget nulla. Curabitur sit amet dolor id turpis malesuada blandit.

A szerző

Budapest, 2008. november 22.

Ez a dokumentum szabad szoftver, szabadon terjeszthető és/vagy módosítható a **GNU Free Documentation License**-ben leírtak szerint.

Minden tőlem származó forráskód szabad szoftver, szabadon terjeszthető és/vagy módosítható a **GNU General Public License 3**-ban leírtak szerint.

Az ARM, a Cortex, a Thumb, az AMBA és a CoreSight az ARM Limited bejegyzett márkaneve.

1. A teleport és időgép felépítése

Ide kerül a `__FIXME__` rövid leírása, specifikációja és blokkvázlata.

Mivel lesz benne CAN, rajzolj egy hálózati topológiát. Legyen benne bridge is (tipikusan az USB-CAN átalakító).

Le kellene írni a blokkok funkcióját, és azt, hogy ezek mivel, hogyan vannak megoldva. Csak nagy vonalakban.

Szépen rávezetni az ARM processzorra.

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. Ezek a tulajdonságok a következők:

- 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).
- 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 MMU¹-t, MPU²-t, FPU³-t, ETM⁴-et, megszakítás-kezelőt terveznek bele az ARM cég mérnökei.
- A csipgyártók még számos perifériával egészítik ki a processzor magot: számlálókkal, kommunikációs eszközökkel, A/D és D/A átalakítók, stb.
- Kiváló szoftveres támogatással rendelkeznek: a GCC fordít ARM processzorra. Az OpenOCD⁵ 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 foglalhatnám össze, hogy a két mag ugyan megegyezik, de a processzorok más-más kiegészítőkkal 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 tervet – intellektuális tulajdont (IP) – adja el).

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

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

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

4 Embedded Trace Macrocell – beágyazott nyomkövető egység, hibakeresésre használatos

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

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

2.1. Történeti áttekintés

Ebben a részben az ARM⁶ 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 Acorn⁷ cégnél. Az első, ténylegesen használható kiadás az **ARM2**-es volt, és 1984-től érhető el a piacon. Ennek még 26 bites volt a PC⁸-je, és mindössze 30.000 tranzisztort tartalmazott (a Motorola 68000⁹-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 AT91SAM7¹⁰,
- 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. Z ARM9-es processzorok ARMv4-es és ARMv5-ös architektúrájúak 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 AT-MEL cég AT91SAM9200-as processzora köré épített demo boardon fel tud BOOT-olni a Linux.

A későbbi ARM9-ek már kivétel nélkül ARMv5 architektúrával készültek. Ezek nagy előnye, hogy képesek DSP¹¹ utasítások futtatására (ha a processzor neve „E” betűt tartalmaz: mindegyik tartalmaz). Ha a processzor

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

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

8 Program Counter – utasításslámláló, az aktuális/következő utasítás címét tartalmazza.

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

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

11 Digital Signal Processor – digitális jelfeldolgozó 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).

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, SIMD¹² 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 iPhone¹³-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 Thumb2¹⁴ 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 licenszt, így a közeljövőben várható, hogy ők is elkezdene 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.

¹² Single Instruction Multiple Data – egyazon utasítás több adaton

¹³ <http://en.wikipedia.org/wiki/IPhone>

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

A technika csúcát 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 OMAP3¹⁵ (Open Multimedia Application Platform – nyílt multimédia platform) processzorban, ami kiválóan alkalmazható multimédiás termékekben (Pandora¹⁶).

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 __FIXME__-ben ARM Cortex-M3-at fogok használni**. Ennek oka egyszerű: rendkívül jól megtervezett magról van szó, másrészt az ST-nek köszönhetően már birtokomban van a csipnek számos példánya.

2.2. Az ARM Cortex-M3 blokkvázlata

2.3. Szoftverfejlesztés ARM processzorra

Az előző részben az ARM processzorok történetével foglalkoztam. Ebben az alfejezetben azt fogom leírni, hogy a kiválasztott ARM Cortex-M3 magot tartalmazó mikrovezérlőkre hogyan lehet szoftvert fejleszteni.

2.3.1. C/C++ fordító fordítása ARM architektúrára

A mai mikrovezérlőket már nem érdemes assembly nyelven programozni, mert annyi utasítást és címzési 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 nyomás 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).

Megfontolás tárgyává téve a dolgot úgy döntöttem, hogy a GNU GCC¹⁷ 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-t¹⁸ is használja, ezért ezt is le kell töltenia <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:

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

¹⁶ [http://en.wikipedia.org/wiki/Pandora_\(console\)](http://en.wikipedia.org/wiki/Pandora_(console))

¹⁷ GNU Compiler Collection – GNU fordítógyűjtemény

¹⁸ Tartalmazza az assemblert, linkert, formátom konvertert, stb.

- Először konfiguráljuk a binutils-ot 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. Neken az a véleményem, hogy a C++ számos olyan szolgáltatást tartalmaz, ami megkönnyíti a munkát: 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
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 (másképpen valószínűleg úgymint 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 CRT¹⁹ (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 *szöveg*nek (.text) nevezték, hiszen az assembly és a gépkód majdnem kölcsönösen egyértelműen megfeleltethető). Ezzel a jól bevált hagyománnyal én sem fogok szakítani.

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

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 kötele-

¹⁹ 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).

ző perifériák (megszakításvezérlő, systick²⁰ 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 progeamot inicializásló, 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	Normal (memória)
SRAM_bitband ²¹	+0 ... +2000000	1 MB	Belső
Perifériák	0x40000000-0x5FFFFFFF	500 MB	Perifériák
Külső RAM	0x60000000-0x9FFFFFFF	1 GB	Normal (memória)
Külső eszközök	0xA0000000-0xDFFFFFFF	1 GB	Külső eszközök
Rendszereszközök	0xE0000000-0xE0FFFFFF	1 MB	Rendszereszközök
Gyártóspecifikus	0xE0FFFFFF-0xFFFFFFFF	500 MB	Gyártóspecifikus

Ezek közül a legfontosabbak a memóriaterületek, mert a perifériák esetében úgyis pointerok segítségével végezzük az elérést. A memóriák közül is inkább a FLASH érdemel több figyelmet, ugyanis az itt található azok 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).

²⁰ Olyan számláló / időzítő, amely segít az operációs rendszerek kialakításában, időalapot szolgáltat az ütemező számára.

²¹ A SRAM egy része bitenként is elérhető, így gyorsíthatók a bitműveletek

A flash további 32 bites értékei a megszakítások belépési pontjaira (függvénypointerek – szintén memória-címek, 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.

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  
};  
  
/* -----  
 * 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 kbájt).

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 Makefile²²-t használok:

```
# -----
# This is the Makefile for ST's STM32 (ARM-Cortex based) Microcontrollers
#
# 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 formware 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 ELF²³ 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:

22 Fordítást automatizáló program (make) „konfigurációs” fájlja

23 Executable and Linuakbe 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 legalsó 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
```

Ha a programozó eszköz FT2232-vel lenne felépítve, akkor a libusb és a libftdi függvénykönyvtárak telepítése után így végezzük a fordítást:

```
cd trunk/
./bootstrap
./configure --enable-ft2232_libftdi
make
sudo make install
```

Aki gyári programozó hardvert használ, így fordítsa az openocd-t (példa):

```
cd trunk/
./bootstrap
./configure --enable-ocd_trace
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 elindul az OpenOCD, RESET-elje a mikrovezérlőt.

- Párhuzamos portos programozót használunk.
- Ami a 0x378-as portcímen található.
- A kábel Wiggler típusú.
(http://wiki.openwrt.org/OpenWrtDocs/Customizing/Hardware/JTAG_Cable)
- A JTAG beállításai (ez egy viszonylag gyors kapcsolatot biztosít).
- A csip egy ARM Cortex-M3 típusú, little-endian eszköz, RESET parancs esetén tartsa HALT állapotban (megállítva).
- Időzítési paraméterek RUN és HALT állapotban.
- A FLASH memória típusa, kezdőcíme, mérete.
- A **working area** (munkaterület) a programozás során pufferként szolgál.

```
#daemon configuration
telnet_port 4444
gdb_port 3333
daemon_startup reset

#interface
interface parport
parport_port 0x378
parport_cable wiggler
jtag_speed 1
jtag_nsrst_delay 10
jtag_ntrst_delay 10

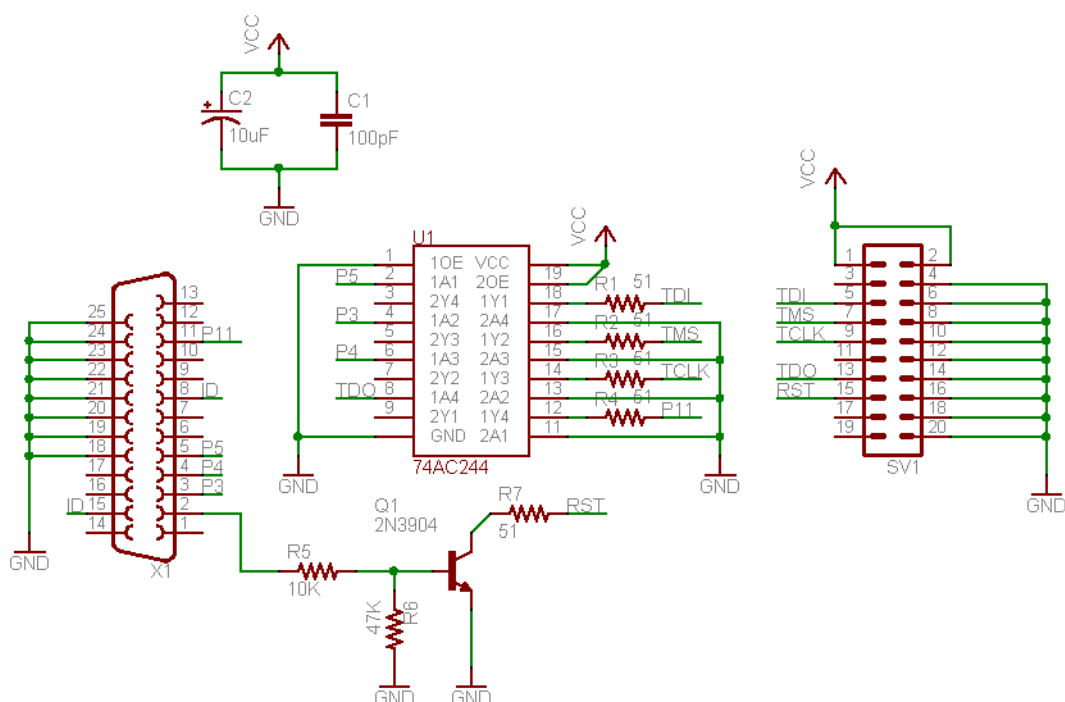
#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst

#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 reset_halt 0
#target_script 0 reset stm32.script
run_and_halt_time 0 30

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

A programozás megkezdése előtt mindenképpen el kell készíteni a JTAG programozó hardvert. Ez egy rendkívül egyszerű áramkör:



Tovább szoktam egyszerűsíteni az áramkört azzal, hogy kihagyom belőle a 74AC244-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 számítógémem `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 tranzistor */

Én 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 tranzistor */
```

A sorok értelmezésére már valószínűleg magától is rájött az Olvasó.

__FIXME__ lecserélni a JTAG képét (google), + Fuszenecker-szabványú 10-pólusú kiosztás + ARM 20-pólusúja is lehetne __FIXME__ breakpoint, új openocd.cfg!.

Olimex, jtag_khz := 1 khz, ha elindult az oszci és a PLL, akkor lehet 500kHz. Jó tartja a frekijét.

Gdb init script

gdb monitor, layout asm, reg, src.

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

```
sudo ./openocd
```

Optimális esetben (néhány óra bibakeresés 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 „flash write_image erase firmware_cortex_m3 0x8000000” paranccsal végezhető.
A helyes válasz:

```
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” majd „resume”

```
> reset
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): 0x9fefffdc (dirty: 0, valid: 1)
(9) r9 (/32): 0xdffdb5fe (dirty: 0, valid: 1)
(10) r10 (/32): 0xa4c3ea69 (dirty: 0, valid: 1)
(11) r11 (/32): 0xc9366b88 (dirty: 0, valid: 1)
(12) r12 (/32): 0xffff7fff (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)
>
```

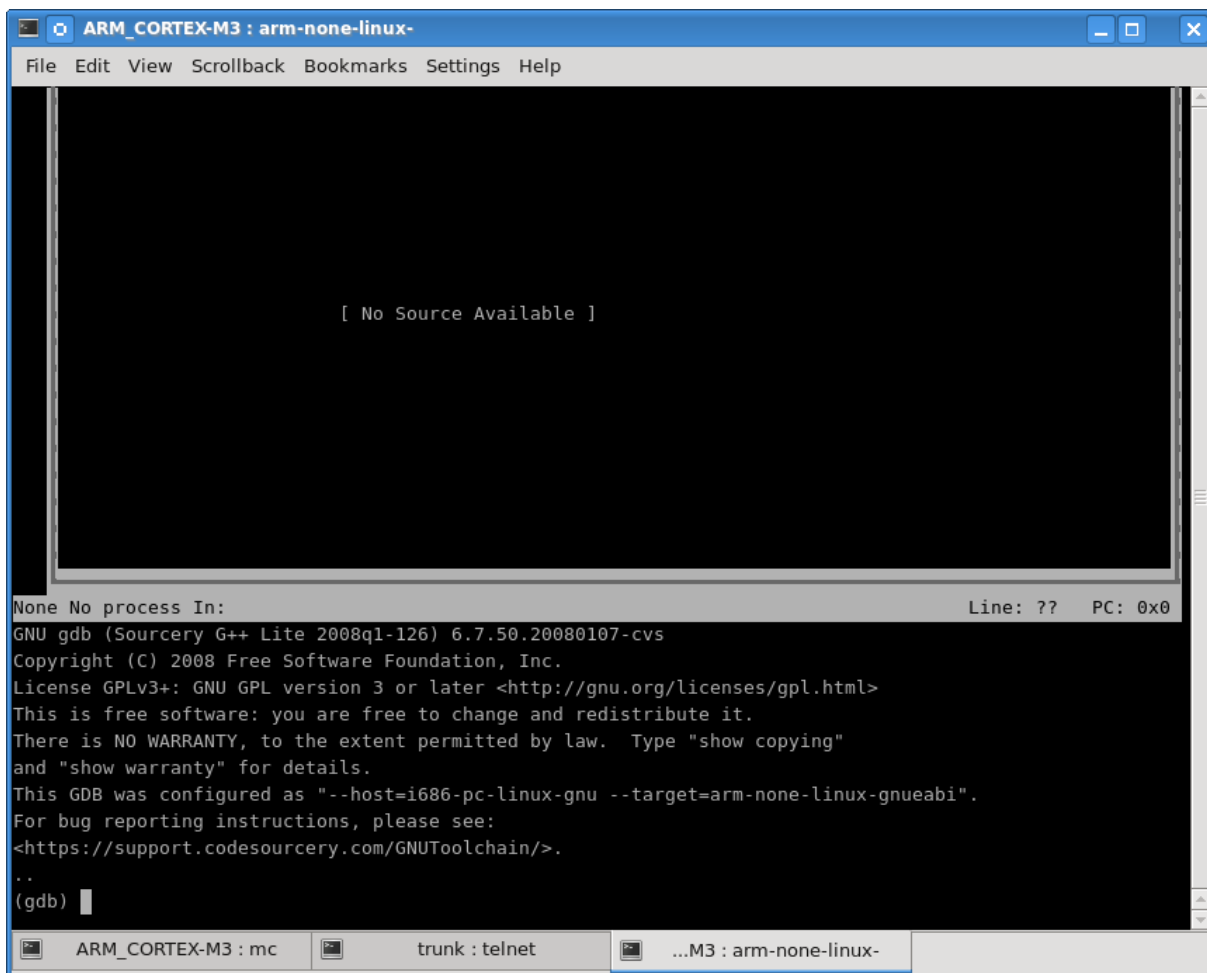
Látható, hogy a mikrovezérlő a 0x5a és a 0x5c című utasításokat hajtja végre. Nem meglepő ez: a végtelen ciklusban ciklál.

```
/* Main loop increments a counter */  
  
for (;;)   
    asm("nop");  
5a:  bf00      nop  
5c:  e7fd      b.n    5a <sysinit+0x1a>
```

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 áramkör).



A „`target remote :3333`” parancssal 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, 0,
27         0, 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 (;;)

```

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)

```

The bottom status bar shows three tabs: `ARM_Cortex-M3 : mc`, `trunk : telnet`, and `...M3 : arm-none-linux-`.

Töréspontot a „b” 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 használandó. 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)

```

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

2.4. Gyártóspecifikus hardver kezelése

STM32 blokkvázlata!

Firmware library, soros port és LED-ek (DEBUG), STM32 specifikus dolgok, PLUSZ a megszakításvezérlő.

3. USB-CAN átalakító

Motiváció: kommunikáció a CAN-es eszközökkel, modern buszok használata, debug.

3.1. A CAN busz működése és használata

A CAN busz tulajdonságai, megvalósítása STM32-ben

Lehet írni a hálózati topológiáról, a hardverről (fizikai réteg, szívásfaktor: földelés, bitsebesség, stb.).

MAC+LLC: Keretek formátuma, maszk, id, priorítás szintek, címek, utasítások.

Itt lehet írni a fw lib. függvényeiről, és a CAN keret küldéséről. Zita tud fényképet csinálni szkópról!

3.2. Kommunikáció a PC-vel: az USB

Ugyanaz, mint a CAN.

3.3. Az USB-CAN átalakító megépítése

A teljes USB-CAN átalakító, blokkvázlattal, kapcsolási rajzzal, NYÁK-tervvel, szoftverrel, szoftver keretrendszer.

4. A teleport és időgép kialakítása

4.1. Motiváció, hardver specifikáció

A __FIXME__ kialakítása, hardveres megfontolások, specifikáció.

4.2. A teleport és időgép felépítése

A __FIXME__ kialakítása, blokkvázlat, kapcsolási rajz, NYÁK-terv, szoftver háttér, szoftver keretrendszer.

4.2.1. Blokkvázlata, működése

4.2.2. Kapcsolási rajza, NYÁK-terve

4.2.3. Szoftver keretrendszer

4.3. Kommunikáció az átalakítóval és más CAN-es eszközzel

kommunikáció a már elmondott CAN-nel (magasabb rétegek?).

Szoftver megoldások.

5. Továbbfejlesztési lehetőségek

ilyen nem lesz.

fejezet:

6. Összefoglalás, végkövetkeztetés

fejezet:

7. Címsor 1

7.1. Címsor 2

7.1.1. Címsor 3

7.1.1.1. Címsor 4

Szöveg

- bullet
- bullet
- bullet
-

7.1.2. Tisztelet Sákjamuni buddhának

fejezet:

8. Felhasznált szoftverek

- OpenOffice.org 2.4 – szövegszerkesztő, szövegformázó
- <http://www.lipsum.com/> – Lorem Ipsum generátor a szövegformázáshoz
- GCC – GNU Compiler Collection (GNU fordítógyűjtemény) – C/C++ fordító
- OpenOCD – Open On-Chip Debugger (nyílt __FIXME__ nyomkövető)
- STM32 Firmware Library – STM32 beágyazott programkönyvtár
-

9. 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)
-

Tárgymutató

A.....	30	purus.....	7
Aliquam.....	7	quam.....	7
átalakító.....	7, 9, 29	rutrum.....	7
dolor.....	7	tellus.....	7
lorem.....	7	tortor.....	7
Praesent.....	7		