SZAKDOLGOZAT FELADAT

Müller Dániel Ágoston
szigorló villamosmérnök hallgató részére

Valós idejű mérések beágyazott rendszerekben JTAG kapcsolaton keresztül

Az autóipari SW fejlesztések szinte kizárólag beágyazott, zárt és egyedi HW eszközökre készülnek. Számos esetben a beágyazott SW ellenőrzése és vizsgálata csakis diagnosztikai szolgáltatásokon keresztül érhető el. Általában DTC (Diagnostic Trouble Code) kiolvasására nyílik lehetőség, ugyanakkor a SW belső vizsgálata, mint pl. állapotgépek működése, jelfolyamok követése, stb., jellemzően nem biztosított.
Az iparágban léteznek meglévő kommunikációs protokollok az ilyen feladatok elvégzéséhez, mint pl. az XCP – Extended Calibration Protocol, ugyanakkor a kiszállított végtermékben ez nincsen támogatva. A feldolgozó egységek JTAG támogatása azonban alkalmas arra, hogy vizsgálat alá vegyük a SW működését és így beágyazott méréseket végezzünk.

A hallgató feladatai:

- Ismerje meg a beágyazott rendszerek valós idejű méréstechnikáját
- Ismerje meg a JTAG interface alkalmazási lehetőségeit
- Készítsen olyan programot, ami futásidőben képes egy megszakítás feletti irányítást átvenni, és minden alkalommal, amikor lefutna az interrupt átirányítja a program vezérlését az egyik saját függvényéhez (kvázi felüldefiniálva az interruptot), ami elvégzi a felhasználó által kiválasztható memóriaterület kiolvasását, és ezt kiküldi JTAG kapcsolaton, amit így naplózni lehet. Ezután adja vissza a vezérlést az interruptnak, hogy a rendszer zavartalanul működhessen tovább.
- A SW eszköz biztosítson lehetőséget a dinamikus paraméterezhetőségre, azaz futásidő alatt meg lehessen adni a vizsgálni kívánt változó memóriacímét és a méretét byte-okban.
- JTAG interfészen a vezérlőre töltve mutassa be és igazolja a SW eszköz működését
- Értékelje ki a naplózás eredményét
- Mutassa be a SW eszköz továbbfejlesztésének lehetőségeit

Tanszéki konzulens: Krébesz Tamás István, tanársegéd
Külső konzulens: Kiss Árpád (Robert Bosch Kft.)

……..
Dr. Dabóczi Tamás
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Müller Dániel

VALÓS IDEJŰ MÉRÉSEK BEÁGYAZOTT RENDSZEREKBEN JTAG KAPCSOLATON KERESZTÜL

KONZULENSEK

Krébesz Tamás István

Kiss Árpád (Robert BOSCH Kft.)

BUDAPEST, 2017

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Müller Dániel**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. 05. 21.

...…….

Müller Dániel

Összefoglaló

A szakdolgozatom keretein belül egy mérőalkalmazást valósítok meg, amely JTAG kapcsolaton keresztül képes valós idejű mérési adatokat olvasni egy alkalmas eszköz memóriájából, anélkül, hogy az eszköz működését érzékelhetően befolyásolná. A kész program egyik alkalmazási területe az autóipari szoftverfejlesztés, ahol elterjedtek a JTAG interfésszel rendelkező beágyazott rendszerek, de gyakran funkcióit tekintve hiányos a rajtuk futó szoftver diagnosztizálására alkalmas eszköz, vagy eleve nincs ilyen eszköz.

Az elkészült alkalmazás működése során első lépésben bekéri a felhasználótól a vizsgálandó memóriacímeket, és az azokról olvasandó byteok számát, majd a mikrokontroller egyik interrupt rutinját helyettesíti a saját mérőfüggvényével. A mérőfüggvény lefuttatja az eredeti interruptot, majd elvégzi a mérést, és a mért adatokat elérhetővé teszi az alkalmazás számára. Az alkalmazás az adatokat kiolvassa, és egy .csv kiterjesztésű naplófájlban tárolja, ami elérhető a felhasználó számára is. A táblázatos naplófájl könnyűszerrel használható az adatok kiértékelésére, pl. grafikon formájában.

Abstract

The purpose of my thesis project is to develop a measurement software, that is capable of reading real-time measured data from the memory of an applicable deivce, without significantly affecting the normal-mode operation of the device. This tool is intended to be applicable to software development in the automotive industry, where embedded systems with a JTAG interface are frequently used, but diagnostic tools for them are often inadaquate, or simply non-existent.

In order to perform the measurement, as a first step, the application enables the user to provide the memory addresses, that they wish to measure, and the number of bytes to be measured from each of them. The program then proceeds to replace one of the repeating interrupt routines of the microcontroller with its own measurement function. This function calls the original routine first (so as not to disturb the intended function), then performs the measurement, and finally loads the measured data into the memory, where the PC application can access it. The application reads, and logs the measurements into a file of .csv format, which can then be used to evaluate the data (eg. by visualization).

1 Bevezetés

Az autóipari szoftverek tesztelése és belső vizsgálata gyakran problémás, hiszen a legtöbb gyártó nem biztosít olyan eszközöket, amelyekkel beható betekintést nyerhetünk a működésükbe.

Beágyazott irányítóegységek viszonylag nagy százalékban tartalmaznak JTAG interfészt, amin keresztül hozzáférhetünk a mikrokontrollerek belsejéhez. Ezt az interfészt megfelelően használva megvalósítható egy olyan szoftver, ami alkalmas bizonyos diagnosztikai, és mérési feladatok elvégzésére anélkül, hogy a gyártó eszközeihez és kódjához hozzáférésünk lenne.

1.1 A feladat részletezése

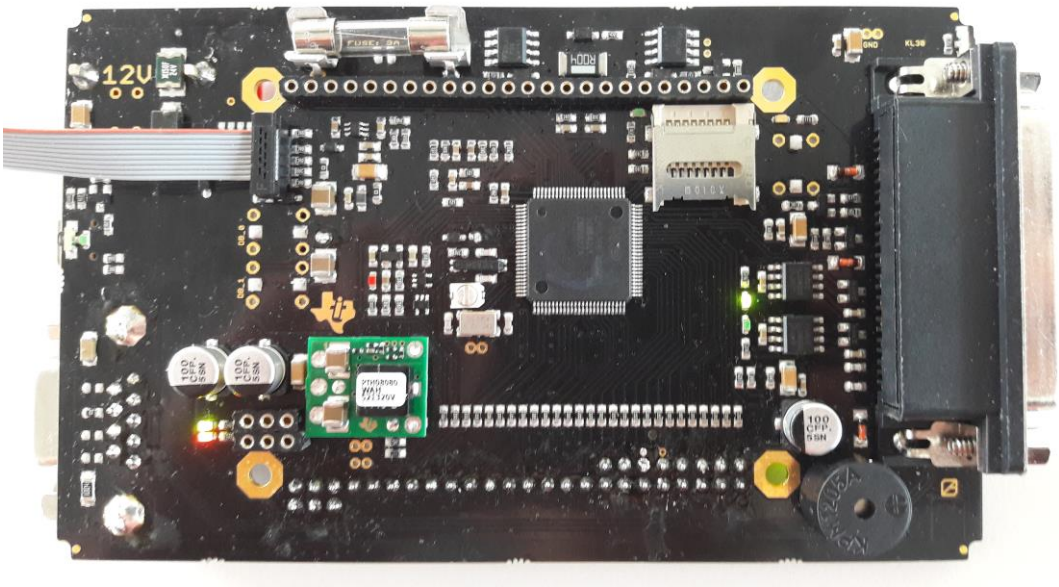
A feladat egy olyan szoftver elkészítése, amely képes JTAG interfészen keresztül felvenni a kapcsolatot egy adott célhardverrel, és átvenni az irányítást a processzor egyik ismétlődő interruptja felett, így teremtve lehetőséget a mérések elvégzésre. A mérés abban merül ki, hogy amikor az interrupt lefutna, helyette a program saját függvénye kerül végrehajtásra, ami először lefuttatja az eredeti interrupt függvényt (így nem veszik el a korábbi funkcionalitás), majd kiolvas a memóriából adott címektől kezdődően adott mennyiségű adatot, és ezt hozzáférhetővé teszi a felhasználó számára. Mind a memóriacímek, mind az azokról kiolvasandó adat paraméterei a programnak, amelyhez a felhasználó hozzáfér, így igény szerint választhatja meg őket.

2 Tervezés és előkészületek

A mérőalkalmazás elkészítéséhez néhány körülmény és előismeret tanulmányozása szükséges. Először is meg kell hozzá ismerni a használandó hardvert, és a kommunikáció fizikai elemeit, majd pedig magát a JTAG szabványt.

2.1 Hardver adottságok

A célhardver egy a BOSCH által fejlesztett CANplayer névvel ellátott eszköz, amely a 2.1 ábrán látható és aminek eredeti feladata, hogy egy CAN (Controller Area Network) hálózatra kapcsolódva emulálni tudja egy valódi autóelektronikai alkatrész működését (pl. motorvezérlő, légzsákkezelő, gyorsulásmérő, stb.) Az eszköz pontos feladata nem lényeges a szakdolgozat szempontjából, elegendő, ha a benne működő processzort ismerjük. A processzor egy ARM Cortex-M3 alapú LPC1769. A processzor minden (a feladat szempontjából) fontos adata megtalálható az NXP által kiadott dokumentációban [1].

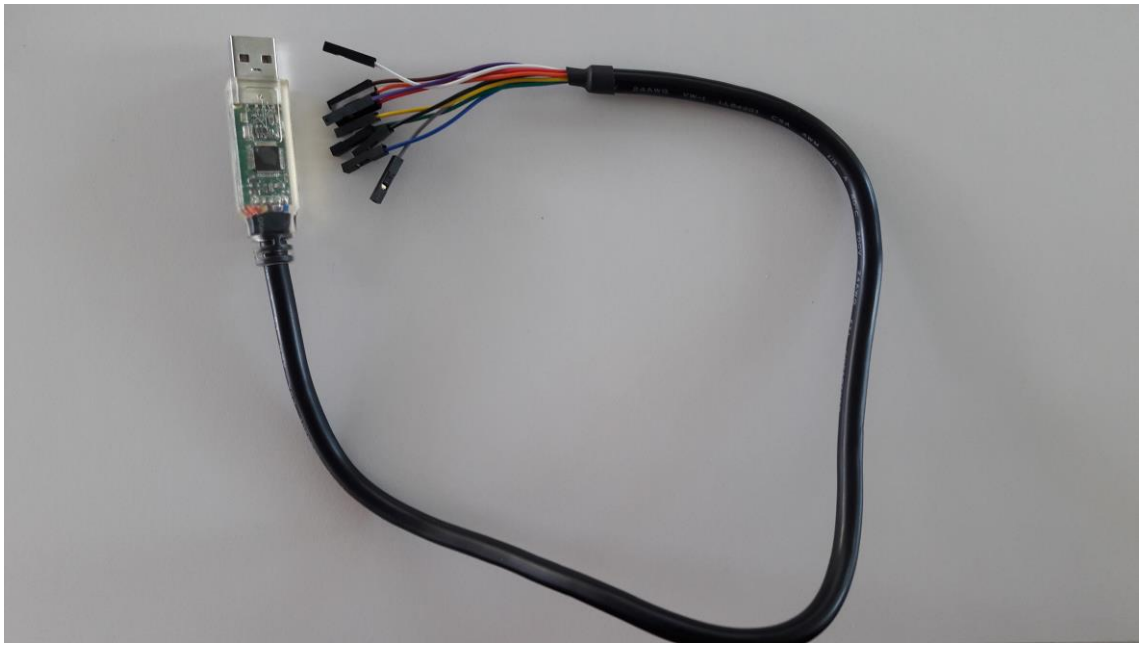


2.1. ábra

CANplayer board LPC1769 processzorral

A cégnél rendelkezésre állt egy FTDI gyártású C232HM kábel [2], amely a 2.2 ábrán látható. Ez tartalmaz egy FT232H IC-t, ami képes USB Hi-speed (480 Mbit/s) sebességen üzemelni, és számos soros protokollra felkonfigurálható, többek között a

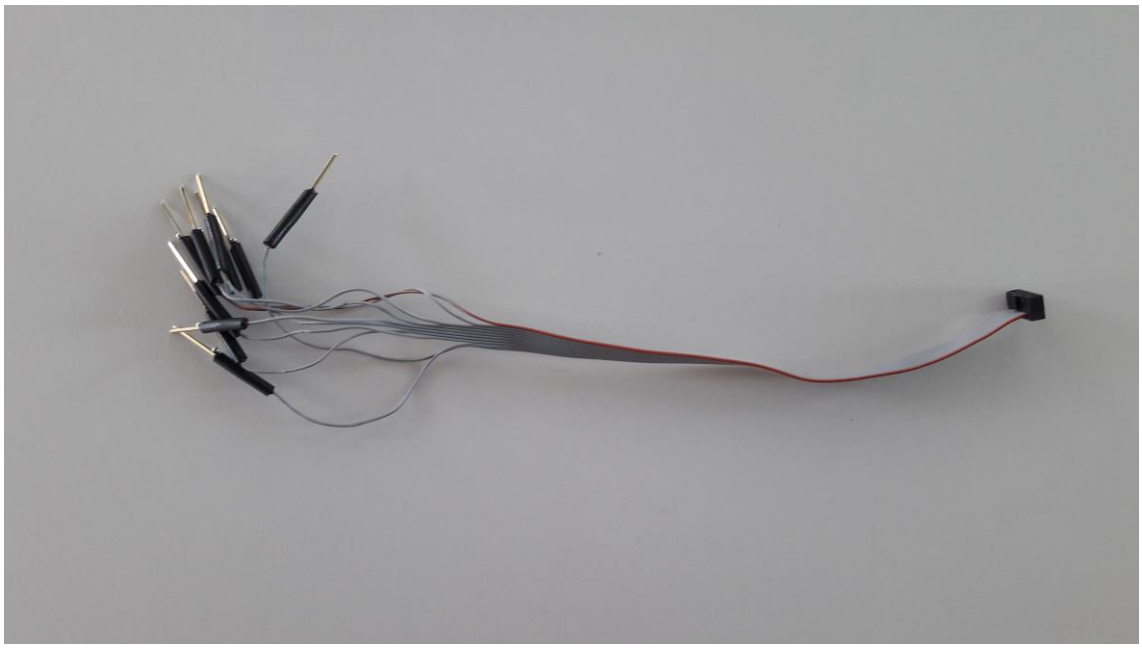
JTAG-re is. Az FTDI honlapján nagyszámú függvénykönyvtár és példakód érhető el, amelyek működnek ezzel az IC-vel, és hasznosnak bizonyulhatnak a kommunikáció felépítésekor. A kábel kimenete egy tízpólusú „octopus” csatlakozó, amely nem teljesen kompatibilis a CANPlayer panelen megtalálható microJTAG tüskesorral. Ez apró kellemetlenséget jelent, hiszen egy köztes átalakító-kábelre van szükség a kapcsolat megteremtéséhez.



2.2. ábra

C232HM kábel

Ezt a kapcsolatot egy 10 eres szalagkábel segített megoldani, amelynek egyik végére egy microJTAG csatlakozó került, másik végét erenként felbontva, és blankolva szabaddá váltak az erek fém részei, amelyek így már forraszthatóak lettek. Tíz tüske került forrasztásra az erek végére, amik már passzoltak az FTDI kábel csatlakozójához. A tartós mechanikai stabilitás érdekében a forrasztásokra zsugorcső került. Az elkészült kábel egyik vége csatlakoztatható lett a panelen található tüskesorra, míg a másik tetszés szerint kapcsolódhatott az octopus csatlakozóhoz (a panelen standard, Cortex Debug csatlakozó található, amelynek az ARM egyik hivatalos dokumentumából [3] kinyerhető a lábkiosztása, a kábel csatlakozójának színkódolása pedig megtalálható a kábel dokumentációjában [2]). Ez a megoldás a fejlesztés idejére elegendő, a későbbiekben azonban célszerű lenne egy elegánsabb megoldást alkalmazni, amely robosztusabb kapcsolatot tud biztosítani.



2.3. ábra
saját készítésű csatlakozó kábel

2.2 A JTAG szabvány

A JTAG (Joint Test Action Group) az IEEE 1149.1 szabványban [4] van részletezve. A szabvány eredeti célja egy olyan metódus kidolgozása volt, amivel egyszerűen ellenőrizhetőek egy nyomtatott áramkör elemeinek fizikai kapcsolatai. Az évek során a JTAG túlnőtt ezen a feladaton, és mára az egyik alapvető elemévé vált a beágyazott rendszerek világának. Ahhoz, hogy a processzorral JTAG-en keresztül tudjunk kommunikálni, elengedhetetlen a szabvány bizonyos részeinek ismerete. A JTAG utasítások kézbesítését egy TAP (Test Access Port) végzi, amelynek hagyományosan legalább a következő négy jelcsatlakozással kell rendelkeznie:

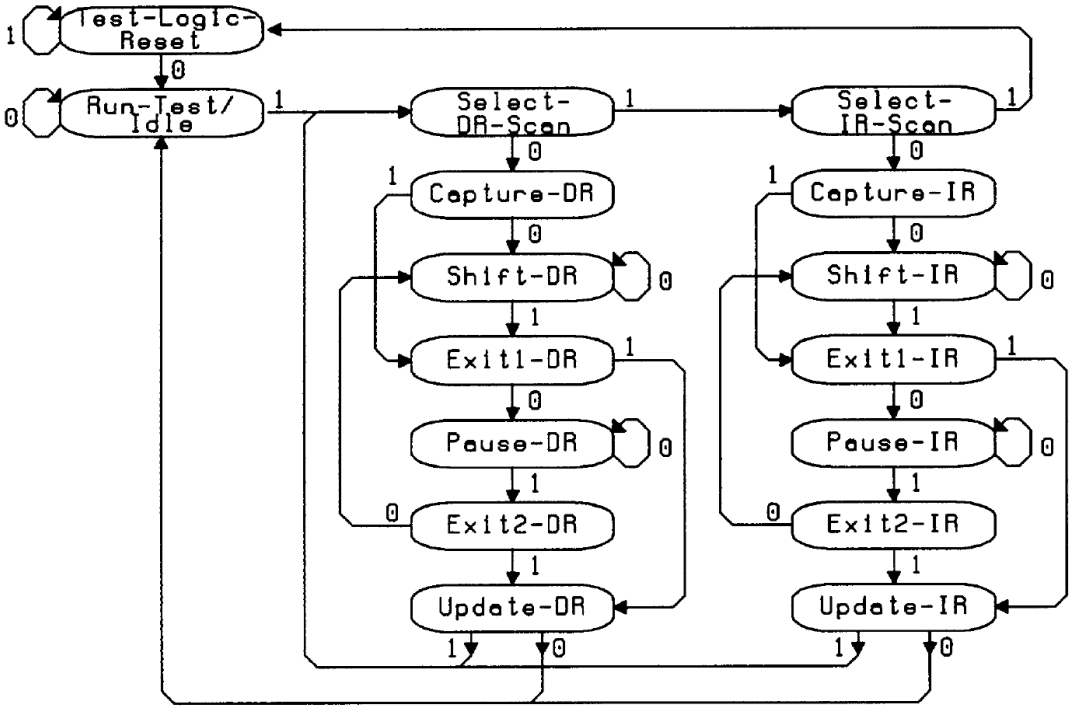
- TDI (Test Data In) – TAP adat bemenete (bemenet)
- TDO (Test Data Out) – TAP adat kimenete (kimenet)
- TMS (Test Mode Select) – a TAP irányító állapotgép vezérléséhez használt jel (bemenet)
- TCK (Test Clock) – órajel (bemenet)

A Cortex-M3 esetében - mint ahogy az ARM processzoroknál általában - ezek kiegészülnek egy RTCK (Return Test Clock) jellel, melynek az adaptív órajel-

generálásnál van szerepe. Ez lehetővé teszi, hogy a debug port visszajelzése alapján a küldő finomhangolni tudja a kommunikáció órajelét.

Egy JTAG eszközzel a TDI és TDO vonalakon tudunk kommunikálni a következőek szerint: Először kiválasztjuk a megfelelő regisztert, amit a TDI és TDO közé szeretnénk illeszteni a TAP állapotgép megfelelő használatával, majd a TCK felfutó éleire egyesével beshifteljük a TDI vonalon a regiszterbe írni kívánt adatokat. Amennyiben olvasni szeretnénk a regiszterből, azt a TDO kimenet olvasásával tehetjük meg. Az olvasást szintén TCK felfutó élére érdemes végrehajtani, hiszen minden változás a TDO vonalon lefutó órajelre történik. Érdemes megjegyezni, hogy nincs akadálya egy regiszter párhuzamosan történő írásának és olvasásának (párhuzamos be- és kishiftelés) sem. A legfontosabb regiszter az Instruction Register (IR), amely tárolja az éppen aktív utasítást. Minden egyéb regisztert (amennyiben nincs külön megnevezve) a Data Register (DR) kollektív jelzővel illetünk. A TDI és TDO közé illesztett regiszter értékei nem azonnal változnak íráskor, mert a TDI és TDO között ténylegesen csak egy ún. scan-chain helyezkedik el (ez lényegében egy shift regiszter), aminek értéke csak az írás befejeztével töltődik bele (párhuzamosan) a kívánt regiszterbe.

2.2.1 A TAP állapotgép



2.4. ábra

TAP állapotgép az IEEE 1149.1 szabvány [4] alapján

A TAP működésének legfontosabb eleme az állapotgép, amely meghatározza, mely regiszter kerül éppen a TDI és TDO közé, és hogy milyen JTAG utasítás van éppen érvényben. Az egyes állapotok közti váltás a TCK felfutó élére történik, a TMS jelenlegi értéke alapján. Az állapotgép működését a ábra 2.4-es ábra írja le.

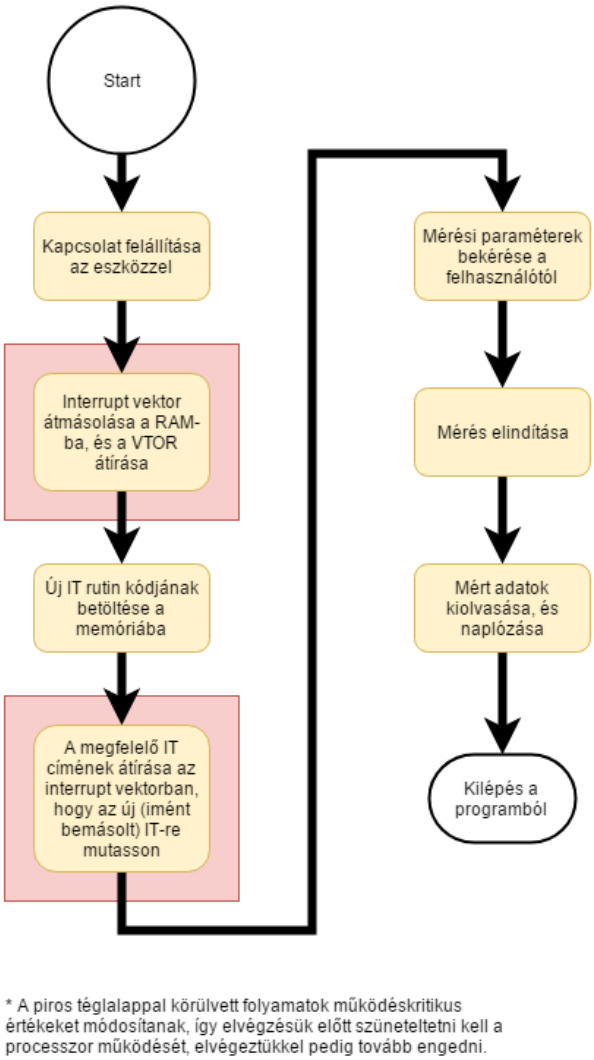
Resetet követően az állapotgép a Test-Logic-Reset (TLR) állapotba kerül. Ekkor az IDCODE (ez az eszköz azonosító kódjának kiolvasását teszi lehetővé) utasítás kerül betöltésre az IR-be, vagy ha az eszköz nem implementálja az IDCODE utasítást, akkor a BYPASS (ez egy 1 bites shift regisztert helyez a TDI és TDO közé) utasítás lesz aktív. Az állapotgép bármikor a TLR állapotba juttatható, ha a TMS legalább 5 órajelen keresztül alacsony logikai szinten van. Az állapotgép egészen addig ebben az állapotban marad, amíg TMS alacsony szinten van, és amint magasnak észleli TMS-t egy TCK felfutó órajelnél, átvált a Run-Test-Idle (RTI) állapotba. Itt a jelenleg aktív utasítástól függően vagy teszt műveletek hajtódnak végre, vagy pedig tétlen marad a teszt rendszer. Az RTI hasonlatos a Select-DR-Scan, a Select-IR-Scan, az Exit1-DR/IR, és az Exit2-DR/IR állapotokhoz abban, hogy nem szolgál más célt, mint hogy köztes állapot legyen két funkcionális állapot között. A Capture-DR állapotban az éppen kiválasztott regiszter értéke párhuzamosan betöltésre kerül a TDI és a TDO közötti scan-chain-be, amennyiben van párhuzamos kimenete, és a jelenlegi utasítás ezt megkívánja. A Capture-IR állapotban a scan-chain-be nem az IR tartalma kerül betöltésre, hanem egy fix érték, aminek az alsó két bitje b01, míg a többi bit a fejlesztő döntése szerint kerül feltöltésre. A Shift-DR állapotban a scan-chain szabadon manipulálható a TDI és TDO segítségével. A TCK felfutó éleire az eszköz beolvassa a TDI-n megjelenő értéket a scan-chain első bitjére, míg a TCK lefutó éleire a TDO kimenetre másolja a scan-chain utolsó bitjének értékét. Mivel a scan-chain egy soros shift-regiszter, így két írás/olvasás között az értékek egyet lépnek a regiszter vége felé. Ilyen módon az egész scan-chain tartalma kicserélhető, vagy kiolvasható a Shift-DR állapotban. Shift-IR állapotban hasonló a működés, csak itt a scan-chain nem egy regiszter jelenlegi értékét tartalmazza az állapot elején, hanem egy fix értéket (ami a Capture-IR állapotban került bele). Elsődleges célja a Shift-IR állapotnak, hogy új utasítást tudjunk az IR scan-chain-jébe másolni. A Pause-DR/IR állapotok segítségével meghatározatlan ideig szüneteltethetjük az írást/olvasást, vagy szünetet tarthatunk két művelet között. Az Update-DR állapotban a DR scan-chain tartalma párhuzamosan betöltésre kerül a DR-be, a TCK lefutó órajelére, amennyiben a jelenlegi utasítás ezt lehetővé teszi. Az Update-IR állapotban

hasonlóképp betöltődik az IR tartalma, de ez utasítástól függetlenül megtörténik. Az újonnan bemásolt utasítás ettől a pillanattól kezdve tekinthető érvényesnek.

2.3 Kivitelezési terv

Miután sikerült a fizikai kapcsolatot megvalósítani, a következő lépés a szoftverelemek működésének megtervezése. Mivel a feladat egy részéhez a processzoron futó kód írása szükséges, míg a másikhoz a PC-n futó kódot kell írni, érdemes ezt a két részt külön kezelni. A PC-s alkalmazás tervezett működését vázlatosan mutatja be a 2.5. ábra. Természetesen az első lépés a kapcsolat felvétele az eszközzel (először az FTDI chippel, majd azon keresztül a mikrokontrollerrel). Ha ez sikeres volt, akkor a következő feladat a processzor egyik interruptjának átirányítása. Az interruptok címei a Cortex-M3 processzorban az NVIC (Nested Vectored Interrupt Controller) részeként, a FLASH memória elején vannak tárolva, amennyiben az aktuális program nem helyezte át. A processzor lehetőséget biztosít az NVIC áthelyezésére, a VTOR (Vector Table Offset Register) értékének módosításával, így akár a FLASH-en kívül is tárolhatjuk az interrupt vektort, ami nagyban megkönnyíti annak módosítását. A VTOR arra is használható, hogy olvasásával meghatározzuk az IT vektor jelenlegi helyét (ez azért szükséges, hogy tudjuk, honnan kell átmásolnunk megfelelő mennyiségű adatot a RAM-ba).

Ahhoz, hogy átírjuk az interrupt vektor címét, először meg kell állítani a processzor futását, hiszen nem kiszámítható az eszköz működése a VTOR átírása során. Miután áthelyeztük az interrupt vektort, módosítjuk az egyik ismétlődő interrupt címét, egy olyan memóriacímre, ahol a saját mérési függvényünk helyezkedik el (az hogy milyen sűrűn ismétlődik az interrupt, a mérések frekvenciáját befolyásolja). Természetesen szükséges lesz az eredeti interrupt függvény címe is, hiszen azt is le kell futtatni, mielőtt a mérési feladatra sor kerülne. A függvénynek, ahova az újonnan átirányított IT mutat, nincs tehát más dolga, mint meghívni az eredeti IT függvényt, majd elvégezni a memóriaolvasás(oka)t, a felhasználó által megadott adatoknak (cím, hossz) megfelelően. Ha a függvény is a helyén van, és az IT vektor is sikeresen át lett helyezve, a processzor zavartalanul folytathatja a működést. A felhasználó ezután úgy tudja majd a paramétereket megadni, és módosítani, hogy egy megadott memóriacímre beírja a kívánt adatokat, az eredményt pedig szintén egy memóriacím olvasásával tudja majd kinyerni.



2.5. ábra

A PC-s alkalmazás működése

A feladat tehát két jól elkülöníthető részre bomlik:

- A PC-n futó alkalmazás, ami a felhasználó számára látható (ez biztosítja a JTAG-en keresztüli kommunikációt, és az adatok kiolvasását, majd naplózását)
- A mikrokontrolleren futó IT rutin, ami a mérést elvégzi, és a mért adatokat elérhetővé teszi a PC-s alkalmazás számára (ennek részletezése a 3.4 fejezetben található)

A fent részletezett terv számos helyen igényli a RAM írását, így elengedhetetlen, hogy előzetes ismereteink legyenek a memória kihasználtságáról. A processzoron futó program írójának felelőssége, hogy biztosítson elegendő mennyiségű, egybefüggő memóriaterületet, és ezen memóriaterület kezdőcímét megadja. A fejlesztés, és a tesztek során 16 kByte memória volt lefoglalva a mikrokontroller memóriájából az alkalmazás számára.

3 A mérőalkalmazás megvalósítása

Jelen fejezetben a feladatot megvalósító mérőalkalmazás tervezése és megvalósítása kerül részletezésre.

3.1 Kommunikáció JTAG interfészen keresztül

A korábban bemutatott FTDI gyártmányú kábel, és átalakító csak elemi szintű JTAG jelek küldésére alkalmas, hogy minél több eszközzel kompatibilis legyen, így a magasabb szintű kommunikáció megvalósítása a fejlesztőre hárul. Az átalakító gyártója az ftd2xx.dll függvénykönyvtárat bocsájtja a chip használóinak rendelkezésére, ami lehetővé teszi az eszköz MPSSE (Multi Purpose Synchronous Serial Engine) módban való használatát. Az MPSSE parancsok [5] alkalmasak a JTAG TAP egyes jeleinek manipulálására (pl. adat beolvasása a TDO vonalról, adat írása a TDI vonalra, adat írása a TMS vonalra stb.). A szakdolgozat során elkészült kód tartalmazza az FTDI által szabad felhasználásra bocsájtott JTAG kommunikációra példát leíró dokumentum [6] egyes részeit amelyek főleg az eszköz inicializálására, és JTAG kommunikációra való felkonfigurálására szolgálnak.

Az MPSSE parancsok az ftd2xx.dll FT_Write() függvényével juttathatóak az FTDI chiphez, ami egyszerre 1 byte adat írására alkalmas. Ennél fogva minden MPSSE parancs, és minden paraméter legfeljebb egy byte hosszú lehet (Pl. több byte írása LSB first sorrenddel, a TCK lefutó élére a 0x19 paranccsal történik, amit először az írni kívánt byteok számának alsó byteja (LengthL), majd a felső byteja (LengthH) követ, végül pedig az írni kívánt byteok, egyesével). Amennyiben a parancs hatására adat beolvasására került sor, ez az adat egy pufferben tárolódik az FT232H-n belül. Az itt tárolt adatot az FT_Read() függvény segítségével olvasható ki, amihez előbb a FT_GetQueueStatus() segítségével le kell kérdezni a pufferben tárolt byteok számát. Ez a három függvény tehát a legfontosabb a feladat érdemi része szempontjából, bár az FT232H eszköz inicializálásához, és szinkronizálásához számos más függvényre is szükség van. Az ftd2xx.dll függvényei egy FT_HANDLE változóval hivatkoznak az FTDI eszközzel való kapcsolatra, amit minden FT függvény megkap paraméterül. Ez az FT_HANDLE egyike azon változóknak, amik elemi részei a kapcsolatnak, így egy struktúrába (jtagConnection) pointerként ágyazva kerül átadásra azon függvényeknek,

amelyek igénylik az eszközzel való kommunikációt. A jtagConnection további elemei egy FT_STATUS pointer, ami az FT függvények visszatérési értékét tárolja el (ez ideális esetben FT_OK), valamint egy saját készítésű TAP_STATE enumáricóra mutató pointer, ami a TAP állapotgép jelenlegi állapotát tárolja. Az alábbi kódrészlet a TAP_STATE enumeráció, és a jtagConnection struktúra definícióit tartalmazza.

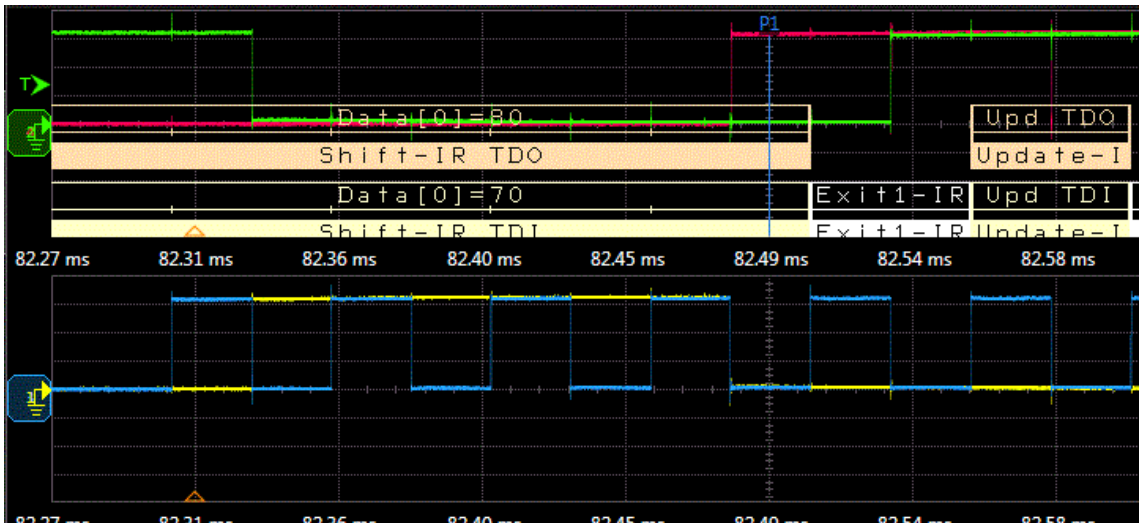
```
enum TAP_STATE {
     TAP_TLR             = 0,
     TAP_RTI             = 1,
     TAP_SELECT_DR_SCAN  = 2,
     TAP_CAPTURE_DR      = 3,
     TAP_SHIFT_DR        = 4,
     TAP_EXIT1_DR        = 5,
     TAP_PAUSE_DR        = 6,
     TAP_EXIT2_DR        = 7,
     TAP_UPDATE_DR       = 8,
     TAP_SELECT_IR_SCAN  = 9,
     TAP_CAPTURE_IR      = 10,
     TAP_SHIFT_IR        = 11,
     TAP_EXIT1_IR        = 12,
     TAP_PAUSE_IR        = 13,
     TAP_EXIT2_IR        = 14,
     TAP_UPDATE_IR       = 15,
     ALL_TAP_STATES      = 16
};

typedef struct {
     FT_HANDLE* ftHandlePtr;
     FT_STATUS* ftStatusPtr;
     TAP_STATE* statePtr;
} jtagConnection;
```

Mivel a struktúra minden eleme pointer, így az azt használó függvények úgy manipulálhatják a változókat, hogy azok értéke a függvényeken kívül is megmaradjon, amire sok esetben szükség is van.

A magas szintű JTAG kommunikációhoz először elengedhetetlen, az egyes TAP jelek megbízható manipulációja. Ezt a feladatot végzik el a következő függvények:

- clockByteIn() – beolvas legfeljebb 8 bitet a TDO vonalról
- clockIn() – beolvas legfeljebb 65536 byteot a TDO vonalról
- clockByteOut() – legfeljebb 8 bitet ír a TDI vonalra
- clockOut() – legfeljebb 65536 byteot ír a TDI vonalra
- step() – 1-et vagy 0-t ír a TMS vonalra, a megadott paraméternek megfelelően, és módosítja a jtagConnection state változóját.

Értelemszerűen az írások és olvasások magukban foglalják a megfelelő órajelek küldését is (pl. 6 bit olvasása a TDO vonalról 6 órajel küldésével is jár). Érdemes külön kitérni az írást végző függvények (clockByteOut() és clockOut()) működésére. Ezek el vannak látva egy extra paraméterrel, amellyel megadható, hogy az adott függvényhívás az utolsó-e, az éppen aktuális írásműveletben. Ha valóban az utolsó, akkor a legutolsó bit küldése külön figyelmet igényel, hiszen ez párhuzamosan kell, hogy történjen az állapotváltó TMS jel küldésével (mindkettő a TCK felfutó élére történik). A 0x4B MPSSE parancs képes ezt a feladatot ellátni, és egyszerre ki tudja küldeni az állapotváltó TMS=1 jelet, valamint az utolsó értékes adatbitet. Mivel az utolsó kiírt bit után szeretnénk az írt adatot érvényre is juttatni, a TMS=1 jelet két órajelig tartjuk fent a paranccsal, így az állapotgép egészen az Update-DR/IR állapotig jut, ahol ténylegesen bemásolódik az adat az írandó DR-be, vagy IR-be. Ilyenkor a step() függvényt megkerülve, „manuálisan” állítódik át a jtagConnection state változója.



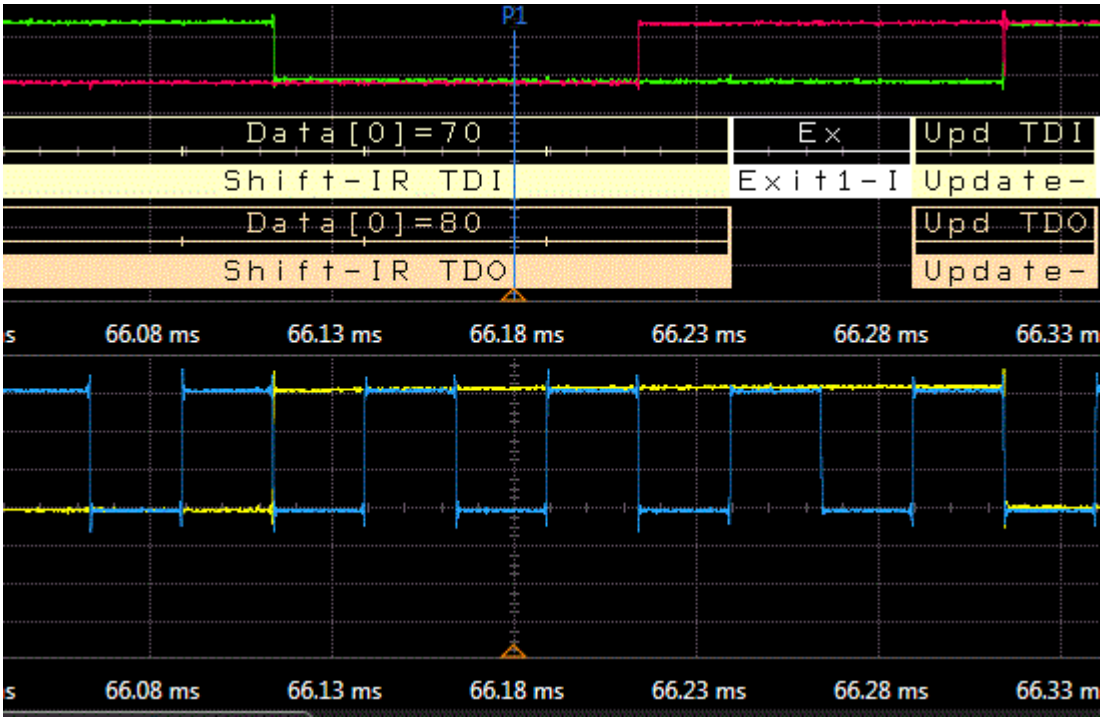
3.1. ábra

IDCODE (b1110) utasítás küldése az állapotváltás figyelembevétele nélkül

Kék jel: TCK; Sárga jel: TDI; Zöld jel: TDO; Piros jel: TMS

A 3.1 ábrán látható, hogy ha nem vesszük figyelembe az utolsó bit külön kezelését, akkor a TDI vonalon nem a megfelelő értékek jelennek meg. A küldeni kívánt adat b1110, amit LSB first sorrendben küldünk, tehát elsőnek egy 0-t akarunk küldeni, utána pedig három 1-est. Ez mind meg is történik, viszont az utolsó bit kiküldésekor még mindig a Shift-IR állapotban vagyunk. Ennek következtében, amikor a következő órajelre megtörténik az állapotváltozás, egy ötödik bit kerül beolvasásra a TDI vonalról, ami mindenképpen 0 lesz. Ez természetesen elrontja az utasításküldést, hiszen a kívánt

b1110 helyett b0111 kerül az IR scan chain-be (az ötödik bit „kitolja” az elsőnek küldött 0-t).



3.2. ábra

IDCODE (b1110) utasítás küldése az állapotváltás figyelembevételével

Kék jel: TCK; Sárga jel: TDI; Zöld jel: TDO; Piros jel: TMS

A 3.2 ábrán már egyszerre történik a TMS jel kiküldése az utolsó bit küldésével, és így a megfelelő adat kerül a TDI vonalra. A TDI magas marad az Exit1-IR és Update-IR állapotokban is (az MPSSE parancs működéséből adódóan), de ennek nincs befolyása a további működésre, mert ezek a parancsok nem olvassák a TDI vonalat.

Ezek a függvények még mindig rendkívül alacsony szinten valósítják meg a JTAG kommunikációt, de legalább elrejtik a fejlesztő elől az MPSSE parancsokat, és letisztultabb rálátást biztosítanak az adat küldésre és fogadásra.

A következő lépcső a TAP állapotgépben való navigáció megvalósítása. A program a korábban már említett jtagConnection struktúrában tárolja a jelenlegi TAP állapotot, amit elsősorban a step() függvény módosít. Az állapotok közti út meghatározását egy routing table [7] (útvonal táblázat) segítségével látja el a goToState() függvény.

	TLR	RTI	SDS	CD	SD	E1D	PD	E2D	UD	SIS	CI	SI	E1I	PI	E2I	UI
TLR	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RTI	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
SDS	1	1	2	1	0	0	0	0	0	1	1	1	1	1	1	1
CD	1	1	1	2	0	1	1	1	1	1	1	1	1	1	1	1
SD	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
E1D	1	1	1	1	0	2	0	0	1	1	1	1	1	1	1	1
PD	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
E2D	1	1	1	1	1	0	0	2	1	1	1	1	1	1	1	1
UD	1	0	1	1	1	1	1	1	2	1	1	1	1	1	1	1
SIS	1	1	1	1	1	1	1	1	1	2	0	0	0	0	0	0
CI	1	1	1	1	1	1	1	1	1	1	2	0	1	1	11	1
SI	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
E1I	1	1	1	1	1	1	1	1	1	1	1	0	2	0	0	1
PI	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
E2I	1	1	1	1	1	1	1	1	1	1	1	0	0	0	2	1
UI	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	2

3.1. táblázat

Útkereső táblázat

A táblázat minden sora, és minden oszlopa egy állapothoz tartozik az állapotgépben, és a sorok és oszlopok indexe megegyezik (pl. a 0. sor és a 0. oszlop is a Test-Logic-Reset állapothoz tartozik). A táblázat egyes elemei 0-k vagy 1-ek, és azt határozzák meg, hogy egy adott állapotból (sor) egy másik állapothoz (oszlop) milyen TMS jel segítségével jutunk közelebb. Az érvénytelen sor-oszlop kombinációk helyén 2-es szerepel. A táblázat tehát nem konkrét útvonalakat tárol, csak a következő lépést a legrövidebb útvonalon. Ha például a Shift-DR állapotból (4-es index) szeretnénk a Run-test/Idle állapotba (1-es index) jutni, akkor a táblázat 4. sorának 1. oszlopában lévő értéket kell vennünk (ez itt 1-es), és azt kell a step() függvénynek átadnunk. A következő lépést már úgy hozzuk meg, hogy a step() függvény által módosított állapotot vesszük kiindulási állapotnak (ez esetünkben Shift-DR helyett már Exit1-DR), míg a cél-állapot változatlan (Run-test/Idle). A táblázat megfelelő cellájának kiolvasását, és a step() függvény meghívását addig ismételjük, amíg el nem érjük a kívánt állapotot.

Ezek után már képesek vagyunk viszonylagos kényelemmel mozogni az állapotgép állapotai közt, és szükség esetén adatot írni az eszköz bemenetére (TDI) és olvasni a kimenetéről (TDO). Tovább egyszerűsítvén a kommunikációt, a sűrűn elvégzett utasítássorozatok ki lettek szervezve magasabb szintű függvényekbe:

- ReadDR() – először elnavigál a Shift-DR állapotba, majd a kívánt mennyiségű adatot kiolvassa a TDO vonalról
- WriteDR() – először elnavigál a Shift-DR állapotba, majd a kívánt adatot kiírja a TDI vonalra, végül elnavigál az Update-DR állapotba (hogy a beírt adat ténylegesen a regiszterbe kerüljön)
- WriteIR() – először elnavigál a Shift-IR állapotba, majd a kívánt utasítást kiírja a TDI vonalra, végül elnavigál az Update-IR állapotba (hogy a beírt utasítás érvényre is jusson)

Ezeken kívül implementálásra került egy navigateToTestLogicReset() függvény is, amely képes TLR állapotba juttatni az állapotgépet, anélkül, hogy tudná, jelenleg milyen állapot van érvényben Ezt a korábban már említett módon, a TMS 5 órajelen keresztüli magas szinten tartásával éri el. Erre azért van szükség, hogy a program elején egy ismert állapotba tudjuk hozni az állapotgépet, mielőtt elkezdünk benne mozogni. A függvény visszatérési értéke felhasználható úgy, mint az állapotváltozó kiinduló értéke (TAP_TLR, ha sikeres volt a TMS jelek küldése, TAP_ALL_STATES, ha nem).

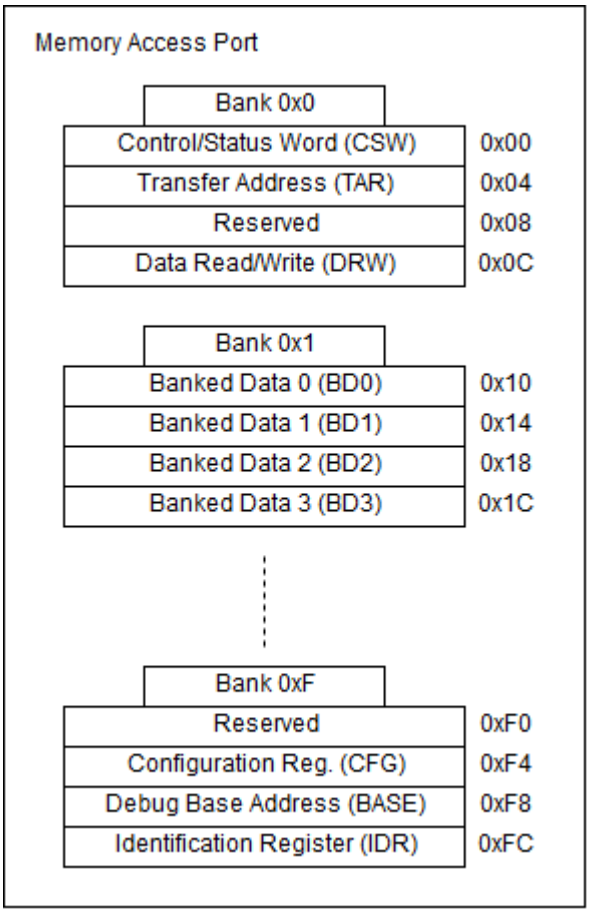
3.2 Hozzáférés a memóriához

A 2.3 pontban leírtaknak megfelelően, az egyik legfontosabb részfeladat a memóriához való hozzáférés. Ide kell átmásolni az IT vektort, itt fog elhelyezkedni a mérést végző függvény, továbbá memóriacímzéssel elérhető néhány processzor-regiszter is, amelyek írására és olvasására szükség lesz a feladat végrehajtásához.

3.2.1 ARM Debug Interface Architecture

A Cortex-M3 processzor része egy sztenderd, CoreSight Debug Port, ami az ARM Debug Interface Architecture [8] (v5.0) specifikációinak megfelel.

A specifikáció szerint a memória elérhető egy MEM-AP-n (Memory Access Porton) keresztül. A specifikáció szerint a MEM-AP a 3.3 ábrán látható módon épül fel.



3.3. ábra

Memory Access Port regiszterek

A MEM-AP CSW (Control Status Word) regiszterének írásával konfigurálható a hozzáférés módja, a TAR (Transfer Address Register) írásával beállítható a hozzáférés (kezdő) memóriacíme, végül a DRW (Data Read Write) regiszterből kiolvasható a címen található adat, és ugyanezen regiszter írásával módosítható a memória tartalma. A működés megértéséhez elengedhetetlen a CSW regiszter (3.2. táblázat) néhány részének ismerete.

31	30 … 24	23	22 … 16	15 … 12	11 … 8	7	6	5 … 4	3	2 … 0
DbgSwEn	Prot	SPIDEN	Reserved	Type#	Mode	TrInProg	DeviceEN	AddrInc	Reserved	Size

3.2. táblázat

CSW regiszter

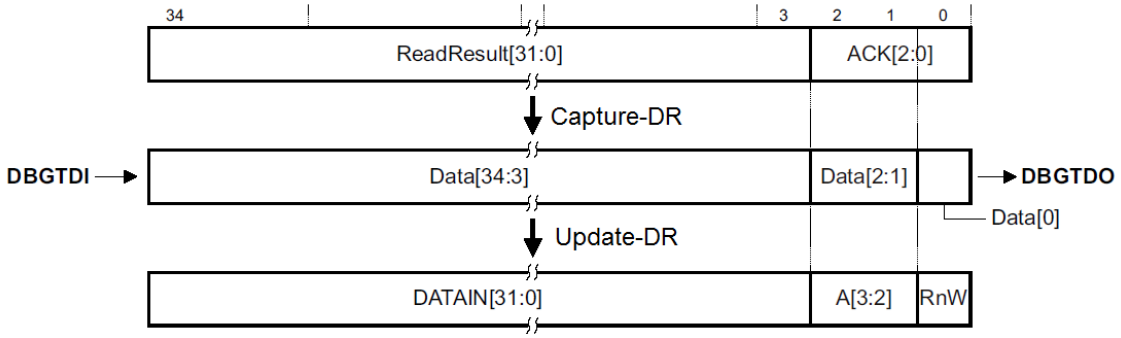
- Size: A hozzáférés egységének mérete (8bit/16bit/32bit/64bit/128bit/256 bit).
- AddrInc: Cím automatikus növelése egyes hozzáférések után. A mező értéke alapján a TAR értéke maradhat változatlan egy írás/olvasás után, vagy pedig növekedhet a hozzáférés méretének megfelelően (pl. 32 bites méret esetén 4-el növekszik a TAR, ha be van kapcsolva az AddrInc).
- Prot: Az itteni bitek szabályozzák a memória hozzáférés engedélyeit. Pl. ha az NVIC regisztereit szeretnénk írni (ilyen pl. a feladathoz szükséges VTOR), akkor ennek a mezőnek kell bizonyos bitjeit egyesre állítanunk.

A MEM-AP-n keresztüli hozzáférés folyamata tehát a következő lépésekből épül fel:

- CSW regiszter megfelelő értékekkel való feltöltése
- TAR regiszter feltöltése az írni/olvasni kívánt címmel
- DRW regiszter írása/olvasása

Amennyiben a CSW regiszterben be van állítva a cím automatikus növelése, az utolsó lépés ismételhető addig, amíg az írni/olvasni kívánt memóriarész végére nem értünk, ha viszont nem növekszik magától a cím, akkor az első lépést leszámítva az egész folyamatot ismételni kell, amennyiben nagyobb memóriaterülethez akarunk egyszerre hozzáférni.

A MEM-AP működése, és használata már tisztázott, viszont az még nem, hogy a JTAG kapcsolat segítségével hogyan érhetőek el a regiszterei. A specifikáció erre is kitér, és megköveteli minden kompatibilis eszköztől, hogy implementálja a DPACC (Debug Port Access) és APACC (Access Port Access) JTAG utasításokat. A DPACC segítségével elérhetőek a DP (Debug Port) regiszterek (ezek közül számunkra a CTRL/STAT, és a SELECT_AP a fontosak), míg az AP (Access Port) segítségével az éppen kiválasztott Access Port regiszterei válnak hozzáférhetővé. A specifikációnak megfelelően mind az AP, mind a DP regiszterek 32 bitesek, de a DPACC és APACC regiszterek ennél 3 bittel hosszabbak, hogy el tudják tárolni az előző hozzáférés eredményességét a 3.4 ábrán látható módon.



3.4. ábra

DPACC és APACC működése (forrás: ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 [8])

A DPACC/APACC olvasása esetén a beolvasott adat a regiszter [34:3] bitjein helyezkedik el, míg az előző hozzáférés ACK kódja a [2:0] biteken. A DPACC/APACC írása esetén az írandó adat a [34:3] bitekre kell, hogy kerüljön, a választott DP/AP regiszter címének felső két bitje a [2:1] bitekre kerül, míg a [0] bit határozza meg, hogy a DP/AP regisztert írni (RnW = 0) vagy olvasni (RnW = 1) akarjuk-e. Az ACK kód két féle értéket vehet fel: lehet OK/FAULT (b010), vagy WAIT (b001). WAIT esetén a hozzáférést meg kell ismételni, míg OK/FAULT esetén a hozzáférés végbement (de ez még nem garantálja a sikerességét). Az egyes DP és AP regiszterek címeit a specifikáció részletezi, ezeket a továbbiakban ismertnek tekintjük.

Amennyiben egy DP regiszterből szeretnénk olvasni, akkor először a WriteIR() függvényt használva beírjuk a DACC utasítást az IR-be, majd a WriteDR() függvénnyel beírjuk az olvasandó regiszter címének felső két bitjét a [2:1] helyekre, és egy 1-est a [0] helyre (ezzel jelezve, hogy olvasni akarunk). Ezután a ReadDR() függvényt használva kiolvashatjuk a regiszter tartalmát. Természetesen csak akkor tudunk meggyőződni arról, hogy az olvasás sikeres volt, ha a következő olvasáskor kinyert ACK kód OK/FAULT, és a CTRL/STAT regiszterben nincsen beállítva egyetlen error flag sem.

Ahhoz, hogy a MEM-AP-t használni tudjuk, először is, meg kell, hogy győződjünk róla, hogy mind a rendszer, mind a debug port működnek, és a MEM-AP van kiválasztva.

31	30	29	28	27	26	25 24	23 12	11 8	7	6	5	4	3 2	1	0
CDBGRSTREQ	CDBGRSTACK	CDBGPWRUPREQ	CDBGPWRUPACK	CSYSPWRUPREQ	CSYSPWRUPACK	Reserved	TRNCNT	MASKLANE	WDATAERR	READOK	STICKYERR	STICKYCMP	TRNMODE	STICKYORUN	ORUNDETECT

3.3. táblázat

CTRL/STAT regiszter

Előbbihez a CTRL/STAT regiszterben (3.3. táblázat) kell a megfelelő biteket (CDBGPWRUPREQ, CSYSPWRUPREQ) egyesre állítanunk, utóbbihoz pedig az AP_SELECT regiszterbe kell a megfelelő értéket bemásolnunk. Esetünkben az AP_SELECT 0x00000000 értéke esetén kerül kiválasztásra a MEM-AP (mivel az az első az AP-k sorában), de könnyedén meggyőződhetünk arról, hogy a megfelelő AP került-e kiválasztásra, ha az APACC segítségével kiolvassuk az adott AP IDR-ét (Identification Register). Teendőink sorrendje tehát a következő:

- DPACC utasítás kiválasztása
- CTRL/STAT regiszter írása, a CDBGPWRUPREQ és CSYSPWRUPREQ bitek egyesre állításával
- CTRL/STAT regiszter ismétlődő olvasása, amíg a CSYSPWRUPACK és CDBGPWRUPACK biteket egyesnek nem olvassuk.
- AP_SELECT regiszter feltöltése a megfelelő értékkel (0x00000000)
- APACC utasítás kiválasztása
- CSW regiszter megfelelő értékekkel való feltöltése
- Memóriaműveletek elvégzése (korábbiakban részletezve)

Így már tehát minden ismert, hiszen sikerült visszavezetni a memória hozzáférés teljes folyamatát egyszerű JTAG műveletekre.

3.2.2 Megvalósítás

Ahhoz, hogy a programkód átlátható, és kompakt maradjon, elengedhetetlen néhány új függvény bevezetése, amik a memória hozzáférés feladatát látják el. Először is biztosítani kell a DP és AP regiszterek egyszerű kezelését, hogy egy 35 bit hosszú szám helyett elég legyen a regiszter címét (és írás esetén a 32 bitnyi adatot) megadni. Ezt a célt szolgálják a következő függvények:

- readAccessRegister() – az adott című DP vagy AP (attól függően, hogy épp milyen JTAG utasítás van érvényben) regiszter olvasását végzi el
- writeAccessRegister() – az adott című DP vagy AP (attól függően, hogy épp milyen JTAG utasítás van érvényben) regiszter írását végzi el

A DPACC és APACC korábban leírt működésének egyik következménye, hogy egy egyszerű írás vagy olvasás még nem elég ahhoz, hogy meggyőződjünk a művelet sikerességéről. Az aktuális művelet ACK kódja a következő olvasás alsó 3 bitjében jelenik meg, amit mindenképp figyelembe kell venni, ha hibamentes tranzakciókra törekszünk. Az ACK kódok ellenőrzését egy-egy állapotgép látja el a readAccessRegsiterSM() és writeAccessRegsiterSM() függvényekben. Az állapotgépnek két állapota van: IDLE és READ_WRITE. A függvénybe való belépéskor READ_WRITE állapotba kerül, ahol először is elvégzi a kívánt (írás/olvasás) műveletet, majd ellenőrző olvasást hajt végre, és értelmezi a kapott ACK kódot. Amennyiben a kód OK/FAULT, az állapotgép átkerül IDLE állapotba, ami azt jelenti, hogy a művelet sikeresen befejeződött. Ha a kapott ACK kód WAIT, az állapotgép a READ_WRITE állapotban marad, és megkíséreli újra végrehajtani a műveletet. A maximális próbálkozások számát egy előre konfigurálható timeout változó szabályozza, így akkor sem akad meg a program végrehajtása, ha valami probléma folytán folyamatos WAIT választ kap az állapotgép. Mivel az OK/FAULT válasz nem feltétlenül jelenti a művelet sikerességét (pl. illetéktelen írási kísérlet védett memóriaterületre sikertelen írást eredményez), a program befejezése előtt kiolvassa CTRL/STAT regiszterből az error flag-eket, és csak akkor tekinti sikeresnek lefutást, ha egyik sem lett beállítva. Megoldható lenne az error flag-ek ellenőrzése minden egyes DPACC/APACC művelet után, de ez nagyban növelné a végrehajtáshoz szükséges időt.

Ezekkel a függvényekkel már megvalósítható a memóriához való hozzáférés. További egyszerűsítés végett implementálásra kerültek a readMemory() és

writeMemory() függvények is, melyek 32 bites egységekben (szó) képesek a memóriát írni vagy olvasni (maximum 1024 szót egy hívás során). Először mindkét függvény felkonfigurálja a CSW regisztert 32 bites hozzáférési méretre, és kikapcsolt auto-címnövelésre (ennek okai a későbbiekben részletezve lesznek), majd feltöltik a TAR regisztert az első címmel. A kettő függvény közül a writeMemory() az összetettebb, hiszen az írás több akadályba ütközhet, mint az olvasás. Először is ellenőrzi, hogy az írni kívánt terület része-e az írható címtartománynak (a program írhatónak tekinti a regisztereket tároló memóriaterületet, valamint a biztosított szabad RAM területet), majd meghívja a writeAccessRegisterSM() függvényt. A writeMemory() egyik paramétere egy logikai változó (readBack), amellyel jelezni tudja a függvény hívója, hogy szeretné a beírt értéket ellenőriztetni az írás után (ez a paraméter kihagyható híváskor, ilyenkor automatikusan TRUE értékűnek számít). Ha ez a paraméter TRUE értékű, akkor a függvény minden írást követően elvégez egy olvasás műveletet is. Amennyiben nem azonos az olvasott érték az írni kívánttal, az írás ismétlésre kerül. Ha automatikusan növekedne a cím minden hozzáférés alkalmával, akkor az ilyen hibás írások megismétlése tönkretenné a folyamatot, hiszen az írni kívánt cím nem változna, viszont a TAR regiszterben lévő érték megnövekedne. Maga az írás és az olvasás a korábban bemutatott writeAccessRegsiterSM() és readAccessRegisterSM() függvények ciklikus hívásával történik, amit a TAR regiszter új címmel való feltöltése követ.

3.3 Az Interrupt vektor áthelyezése

A következő lépés az interrupt átirányítása, amihez az interrupt vektor egyik elemét kell módosítani. Az NVIC alapértelmezetten a FLASH memóriában helyezkedik el, amit hagyományos módon nem tudunk a MEM-AP-n keresztül írni. A Cortex-M3 viszont lehetőséget biztosít az NVIC áthelyezésére (remapping), amivel akár a RAM-ba is átköltöztethető az egész NVIC, ahol már szabadon írhatjuk át az elemeinek értékét. Ehhez először meg kell határozni az új helyét az interrupt vektornak. Az ARM hivatalos dokumentációját [9] követve meghatározható az érvényes címek halmaza. Attól függően, hogy legfeljebb hány interrupt tárolható a vektorban, más határokra helyezhető a vektor kezdőpontja. Az interruptok számát fel kell kerekíteni a következő kettő hatványra, majd a kapott számot meg kell szorozni 4-el. Esetünkben az interrupt vektor hossza 200 byte [1], és minden 4 byte egy-egy interruptnak ad helyet, tehát összesen 50 interruptal kell számolnunk. A következő kettő hatvány a 64, aminek a négyszerese 256, tehát a vektor kezdőcíme 256 egyik többszöröse kell, hogy legyen (ami megfelel annak,

hogy az alsó 7 bitje 0). Ennek a számnak az egyetlen függvénye az interruptok száma, így könnyedén átalakítható egy függvénnyé. A programban megadható 200-tól eltérő IT vektor hossz, amit ez a függvény automatikusan beleszámít a képletbe.

Ha már megvan a használható címek halmaza, akkor megkereshetjük az első ilyen címet a használható RAM tartományunkon belül, ahova majd bemásolhatjuk az IT vektort. Alapértelmezetten az NVIC a FLASH legelején (0x00000000) kezdődik, de a program képes alkalmazkodni ahhoz is, hogy ha az IT vektor már eleve át lett helyezve. A Cortex-M3 egyik regisztere, a VTOR [9] (Vector Table Offset Register) azt tárolja, hogy melyik memóriacímen kezdődik az NVIC. Ezt a regisztert olvasva megkapjuk az IT vektor jelenlegi kezdőcímét. Erről a címről kell kiolvasni az NVIC hosszának (200 byte) megfelelő adatot, és átmásolni a korábban meghatározott címre a RAM-ban. Ezek után még nem érvényesek az újonnan bemásolt interruptok, hiszen a VTOR még mindig a korábbi címre mutat.

Ahhoz, hogy a VTOR értékét módosítsuk, előbb meg kell állítani a processzort, mert nem lehetünk biztosak benne, hogy a futó program épp nem használja valamelyik interruptot. A processzor megállítására is biztosít lehetőséget a Cortex-M3 egyik regisztere, a DHCSR [10]. A DHCSR mezői a 3.4 táblázatban szerepelnek.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Write	DBGKEY																Reserved										C_SNAPSTALL	Reserved	C_MASKINTS	C_STEP	C_HALT	C_DEBUGEN
Read	Reserved						S_RESET_ST	S_RETIRE_ST	Reserved				S_LOCKUP	S_SLEEP	S_HALT	S_REGRDY																	

3.4 táblázat

DHCSR (Debug Halt Control and Status Register)

A DHCSR írásakor a felső 16 bitnek mindenképpen DBGKEY-nek kell lenni, ami 0xA05F, különben a többi beírt bit érvénytelen. A számunkra érdekes mezők a C_DEBUGEN és a C_HALT. A C_DEBUGEN engedélyezi a többi mező írását a debug port részéről, míg a C_HALT-tal megállítható a processzor.

A végrehajtandó lépések tehát a következők:

- Processzor felfüggesztése a DHCSR írásával (C_DEBUGEN = 1; C_HALT = 1)
- VTOR regiszter írása, hogy a korábban meghatározott memóriacímet tartalmazza
- Processzor elengedése a DHCSR írásával (C_DEBUGEN = 0; C_HALT = 0)

A processzor jelenlegi állapota (fut/nem fut) könnyen ellenőrizhető, ha ismerünk egy olyan memóriacímet, amit a futó program gyakran változtat. A fejlesztés idejére egy olyan program lett az eszközre töltve, ami egy 32 bites változót folyamatosan növel. A fordító által generált symbol list fájlból kinyerhető a változó memóriacíme, és ennek többszöri olvasásával meghatározható, hogy fut-e a processzor, vagy nem (ha kellően sok olvasás után is ugyanaz a memóriacímen található érték, akkor a processzor áll, különben pedig fut).

Ha ezt a három lépést sikeresen végrehajtottuk, akkor elértük, hogy az interrupt vektor írható memóriaterületen helyezkedjen el, és mindeközben csak minimális ideig kellett szüneteltetni az éppen futó programot. A következő lépés annak a függvénynek a megírása, ami helyettesíteni fogja az egyik Timer interruptot, és ezen függvény bemásolása a fenntartott memóriaterületre.

3.4 Az interrupt rutin megvalósítása

A mikrokontroller kiszemelt IT rutinját helyettesítő függvénynek a feladata, hogy elvégezze a felhasználó által megadott címekről a kívánt számú byte kiolvasását. Ezt a függvényt is a mérőprogram számára biztosított memóriaterületen kell elhelyezni, és a megfelelő bejegyzést az NVIC-ben át kell irányítani a kezdőcímére. Mielőtt bármit is csinálna a rutin, először meg kell hívnia azt az IT rutint, amelyet helyettesít. Ennek kezdőcímét a PC-s alkalmazás feladata elhelyezni a megfelelő memóriacímre, ahonnan ki tudjuk olvasni, és meghívni, így a lehető legkisebb mértékben sérülnek a mikrokontrolleren futó program időzítési feltételei. Ezek után már szabadon elvégezheti a mérést, aminek a paramétereit a memóriából tudja kiolvasni, ahova a PC-s alkalmazás a felhasználó beállításainak megfelelően elhelyezte. A paraméterek egy táblázat formájában kerülnek a memóriába, amelynek minden sora két (32 bites) elemből áll: az

olvasás kezdőcíme, és az olvasandó byte-ok száma. Egy ilyen táblázatra mutat be egy példát a 3.5. táblázat

	Cím (32 bit)	**Byte-ok száma (32 bit)**
1. mérendő változó	`0x100043AB`	8
2. mérendő változó	`0x10003F20`	32
⋮		
N. mérendő változó	`0x100010E7`	4

3.5. táblázat

Példa egy mérés paramétereit tartalmazó táblázatra

Természetesen a táblázat mellett a táblázat hosszát (N) is el kell helyezni a memóriában, különben a mérést végző IT rutin nem tudná, hogy hol van vége (ennek egy alternatív megoldása lehetne, hogy a táblázat utolsó bejegyzése valamilyen bitminta, ami alapján megállapítható, hogy a táblázatnak vége van). Mivel a mérést egy interrupt keretein belül hajtjuk végre, káros következményei lehetnek az időzítésre nézve, ha túl sok olvasást kell végrehajtani egy-egy mérés során, így a mérésenként olvasható byte-ok maximális száma korlátozott (ez a PC-s alkalmazásban kerül ellenőrzésre).

A memóriaterületek elérése nagyságrendekkel egyszerűbb a rutin keretein belül, hiszen ezúttal olyan programkódban vagyunk, amit közvetlenül a mikrokontroller futtat. Elegendő egy megfelelő pointert az írni/olvasni kívánt memóriacímre állítani, és azzal dolgozni. A rutin addig nem kezdi meg a mérést, amíg a PC-s alkalmazás nem jelezte, hogy készen áll, azaz elhelyezte a paramétereket a memóriában, és felkészült a mért adatok kiolvasására. Ehhez először az IT rutin állítja be a saját mérési indikátorát, amely egy 32 bites bitminta a fenntartott memóriaterület meghatározott címén, majd ha ezt a PC-s alkalmazás sikeresen kiolvasta, ő is beállítja a sajátját (hasonló bitminta egy másik címen). A fent említett pointerek segítségével tehát a mérés paraméterei egyszerűen kiolvashatóak, és hasonlóképpen maga a mérés is elvégezhető. A rutin egyetlen hátralévő teendője a mért adatok elmentése. Erre a feladatra két lehetséges megoldást részleteznek az alábbi 3.4.1 és 3.4.2 alfejezetek.

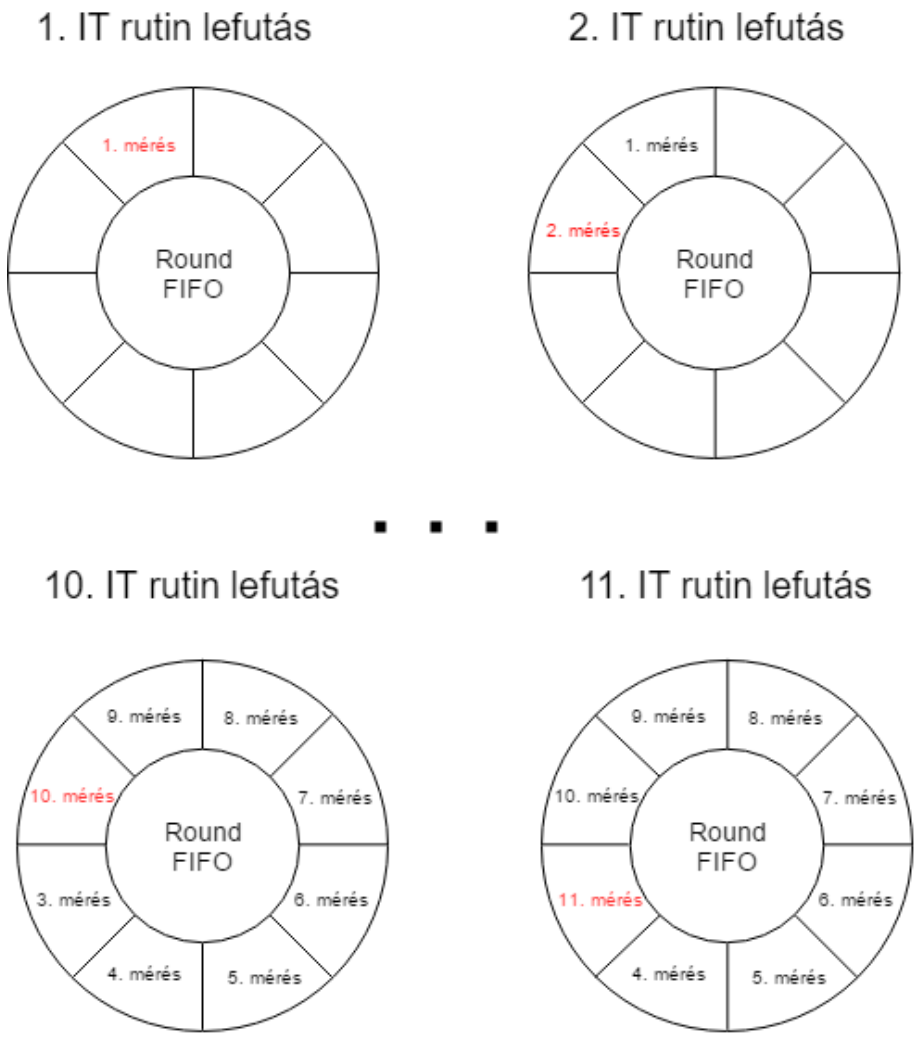
3.4.1 Mért adatok tárolása RoundFIFO-ban

A mért adatok RoundFIFO-ban történő tárolása lehetővé teszi a mérés folyamatos futását, minek során a méréssel párhuzamosan, egyfolytában olvassa a mért adatokat a PC-s alkalmazás.

Miután az IT rutin a mérést elvégezte, a mért adatokat a fenntartott memóriába menti, amelynek része egy roundFIFO szervezésű memóriaterület. A roundFIFO egyes elemei tartalmazzák a mérés sorszámát, amely minden egymást követő mérés hatására inkrementálódik, illetve, a kiolvasott adatokat, abban a sorrendben, ahogy a mérési táblázatban (egy ilyen táblázat lehet mondjuk a 3.5. táblázatban bemutatott példa) meg voltak adva. A PC-s program a mérés sorszámából meg tudja határozni, hogy az aktuális mérés számára fontos-e (nem olvasta-e már ki egy korábbi alkalommal), az utána következő adathalmazt összevetve a táblázattal pedig ki tudja nyerni az egyes mérések értékes adatait.

Azért van szükség a roundFIFO-ra, mert a PC-s alkalmazás nem tudja olyan sűrűn pollozni a mikrokontroller memóriáját, mint amilyen sűrűn a mérések megtörténnek. Éppen ezért a roundFIFO méretét is úgy kell megszabni, hogy két pollozás közt semmiképp ne érjen körbe, hiszen az azt jelentené, hogy a régebbi (de még kiolvasatlan) mérési adatokat felülírták az újak. Ebből következően a RoundFIFO alapú tárolás egyetlen, és egyben kritikus követelménye, hogy nagyságrendileg összemérhető legyen a mérés sebessége a PC-s alkalmazás olvasási sebességével. Ha túl gyors a mérés, vagy túl lassan olvas a PC, akkor a RoundFIFO megtelik, és elvesznek mérési adatok.

A 3.5 ábrán egy nyolc férőhelyes RoundFIFO működése látható. Az IT rutin minden mérés után a következő cellába tölti be a mért adatokat. Ha egyszer körbeért a RoundFIFO-n (ez esetünkben 8 mérés után történik meg), akkor felülírja az első cella tartalmát, és így tovább. A működés akkor megfelelő, ha ekkorra a PC-s alkalmazás már kiolvasta az első cella tartalmát.



3.5. ábra

Egy nyolc férőhelyes roundFIFO működése (pirossal jelölve az aktuális IT rutin lefutása során beírt adatok)

3.4.2 Mért adatok tárolása sorfolytonosan

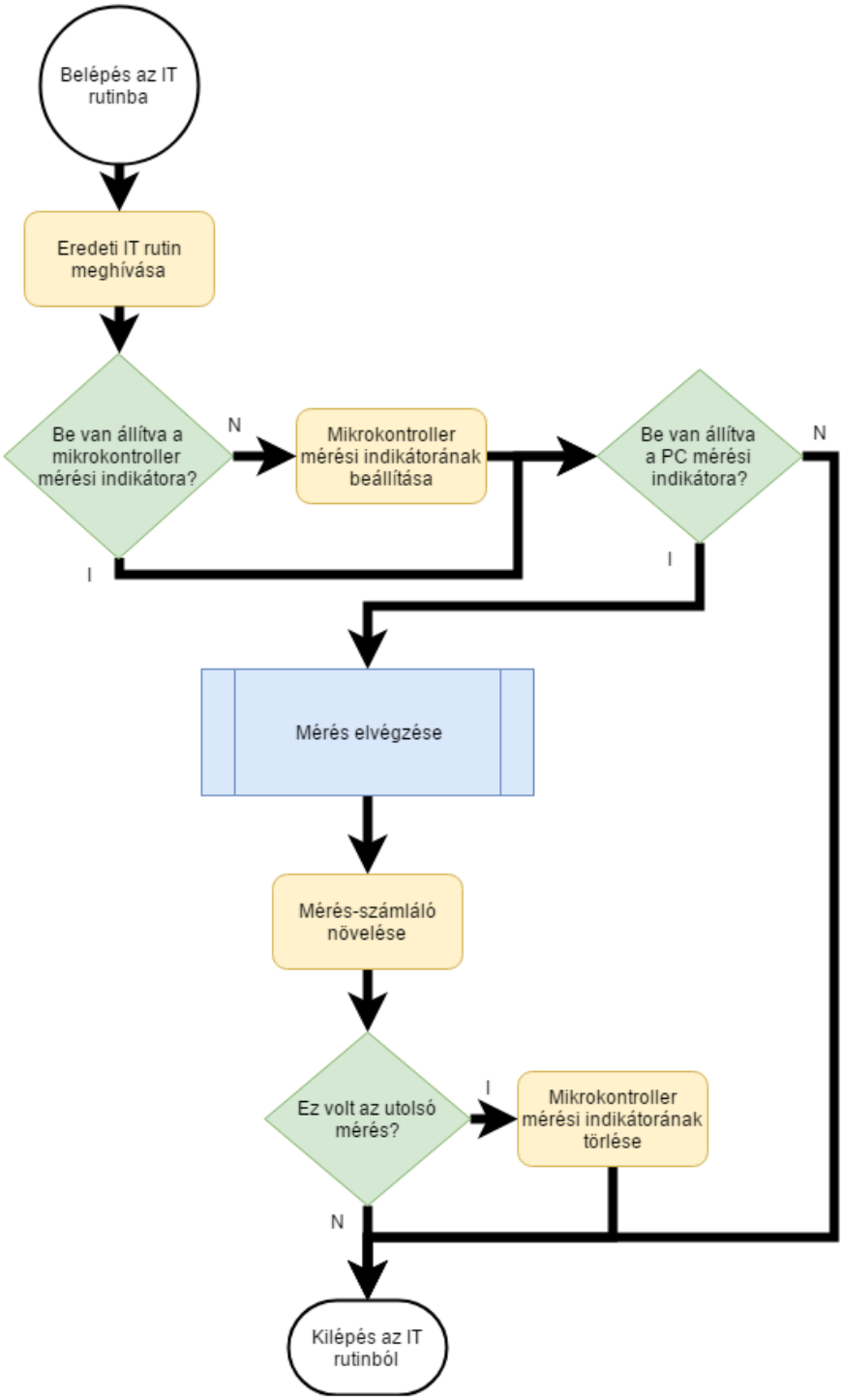
A RoundFIFO-nál egyszerűbb, és biztonságosabb megoldás egy fix méretű memóriaterület fenntartása, amelyben nem íródnak felül korábbi bejegyzések, hanem az új mérések sorfolytonosan követik egymást. Ennek természetesen megvan az a hátránya, hogy nem tarthat tetszőleges ideig a mérés, csak ameddig a memóriaterület mérete engedi. Amit cserébe nyerünk, az a sebességmegkötések elhagyása, és egy konzisztensen alkalmazható tárolási módszer. Míg a RoundFIFO esetében az is befolyásolja a követelmény teljesülését, hogy milyen sűrűn érkeznek az interruptok (így előfordulhat, hogy két, ugyanazon a processzoron futó programból csak az egyikkel használható a mérőalkalmazás), addig a sorfolytonos megoldás teljesen független az interruptok frekvenciájától.

Egyik lehetséges továbbfejlesztése ennek a megoldásnak, hogy kombináljuk a RoundFIFO-val. Ez annyit jelent, hogy amikor elérjük a memóriaterület végét, elkezdjük felülírni az elejét. A PC-s alkalmazás már az után elkezdte kiolvasni az adatokat, hogy az első mérés megtörtént, így a mérés ideje alatt kiolvasott adatok már nem kell, hogy érintetlenek maradjanak. Ezzel kiterjeszthető a mérés ideje, bár a RoundFIFO által biztosított tetszőleges időtartamot még mindig nem közelíti meg.

3.4.3 Választott megoldás

A megvalósított PC-s alkalmazás olvasási sebessége nem megfelelő a RoundFIFO-s megoldáshoz, amennyiben valóban valós idejű méréseket akarunk végezni (ms-os nagyságrendű IT periódusidő). A memóriaolvasások és írások biztonsági okokból ellenőrzésekkel vannak ellátva, amelyek lassítják az amúgy is sok lépcsőből álló folyamatot. Míg a mérést végző IT rutinnak elég egy pointeren keresztül a memóriába írni, a PC-s alkalmazás rá van szorulva a JTAG interfészre, és a TAP állapotgépben való lépkedésre. Ebből következően a sorfolytonos tárolás marad, mint egyetlen lehetséges megoldás. Mivel ez a megoldás csak fix számú mérést engedélyez, valahogy jeleznie kell az IT rutinnak, hogy vége a mérésnek. Erre tökéletesen megfelel a korábban említett mérési indikátor, amit a mérés befejeztével kinullázhat a rutin.

Az IT rutin vázlatos működése látható a 3.6. ábrán. Ebből tisztán látható a folyamatok sorrendje. Először az eredeti interrupt kerül meghívásra, majd pedig az indikátorok beállítása és kiolvasása történik meg. Ha mindkét indikátor be van állítva, akkor a rutin elvégzi a mérést, és a 3.4.2. fejezetben leírtak szerint eltárolja a mért adatokat. A mérés végeztével a rutin inkrementálja a mérés-számlálót, ami a memóriában helyezkedik el, nem pedig a stack-en, így értéke megmarad az IT rutin két lefutása közt. Ha ez a mérés-számláló elérte a felhasználó által végezni kívánt mérések számát, a rutin törli a mérési indikátorát, jelezve ezzel, hogy a mérés befejeződött.



3.6. ábra

Az IT rutin működése

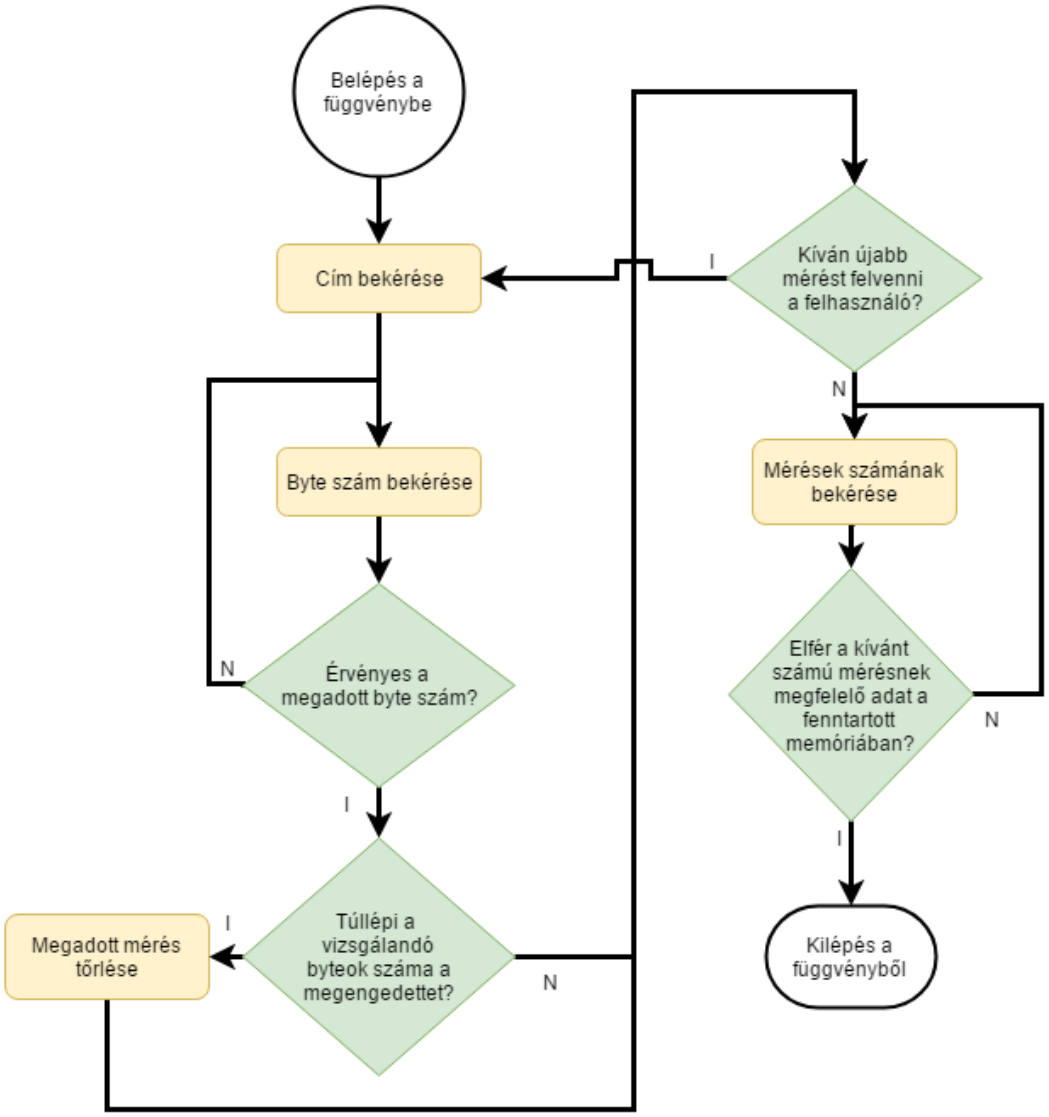
3.5 A mérés előkészítése

A működő IT rutin önmagában még nem elég a mérés elvégzéséhez, hiszen a mikrokontrollerrel is tudatni kell, hogy melyik interrupt helyett hívja meg. Ehhez először el kell helyezni az újonnan megírt rutint a memóriában, majd pedig átállítani az

NVIC egyik megfelelő címét a rutin kezdőcímére. Az IT rutin fordításakor generált bináris fájl tökéletesen megfelel a feladatnak, hiszen szinte kizárólag a rutinhoz tartozó gépi kódot tartalmazza. Ezt a bináris fájlt tehát a PC-s program be tudja olvasni, és tartalmát a korábban részletezett memóriaműveletek segítségével elhelyezheti a memóriában. Ezt követően már át lehet irányítani a kívánt NVIC bejegyzést az új rutin kezdőcímére. Alkalmas bármely olyan NVIC bejegyzés, amely rendszeres időközönként meghívódó IT rutinokra mutat (ilyenek a Timer IT-k, és Cortex-M3 esetében a RIT – Repetitive Interrupt Timer, valamint a SYSTICK – System Tick). Az, hogy a PC-s alkalmazás melyiket fogja átirányítani, egy egyszerű paraméter megváltoztatásával állítható. Mielőtt végleg felülírná a kívánt NVIC bejegyzést, a program elmenti az eredetileg ott található értéket, hiszen ez a címe az eredeti IT rutinnak, amire később még szükség lesz (az új IT rutin erről a címről fogja majd meghívni a régit, mielőtt elvégezné a mérést). Természetesen ezúttal is meg kell állítani a rendszer működését, mielőtt az átírás végbemenne, különben előfordulhat, hogy épp akkor hívódik meg az interrupt, amikor az NVIC-ben található címe tranziens állapotban van. Működő megoldás lenne az is, ha egyetlen egyszer állítanánk csak le a rendszert, és addig nem indítanánk újra, amíg minden kritikus lépést végre nem hajtottunk, de ezzel fölöslegesen sokáig szüneteltethetnénk a végrehajtást, mert a kritikus lépések nem közvetlenül követik egymást. Mindezek után az átirányítás kész, és a rendszer már az új interrupt rutint fogja meghívni a régi helyett. Mivel azonban a mérést csak akkor végzi el az interrupt, ha a PC már beállította a mérést jelző indikátorát, ezért nem csinál egyebet, mint meghívja a régi interruptot, és visszatér.

Mivel a bináris fájl mérete jelentős lehet, érdemes megspórolni annak beolvasását és memóriába helyezését, ha már nincs rá szükség. Amennyiben a PC-s program úgy érzékeli, hogy az NVIC-ben átírandó bejegyzés már a megfelelő helyre mutat (ami egyben azt is jelenti, hogy az új interrupt van érvényben a régi helyett), akkor sem az NVIC bejegyzés lecserélését, sem a bináris fájl betöltését nem hajtja végre. Ellenkező esetben még az is előfordulhatna, hogy éppen akkor írja (újra) az interruptot, amikor az éppen fut, aminek súlyos következményei is lehetnek. Értelemszerűen a betöltés akkor hagyható el, ha nem történt reset a mikrokontrolleren a PC-s program előző lefutása óta, hiszen ekkor a RAM nem került kiürítésre (azt is feltételezzük, hogy a fenntartott memóriaterület valóban fenn van tartva, és nem lett illetéktelenül módosítva).

A következő lépése a mérés előkészítésének, hogy a program bekéri a felhasználótól a mérési adatokat. Ennek folyamatát vázolja a 3.7. ábra. Az alkalmazás jelenleg csak konzolos formában működik, ami valamelyest limitálja a felhasználói beavatkozás összetettségét, de a feladathoz elegendő. Az alkalmazás először egy cím megadását kéri, majd az onnan olvasandó byte-ok számát. A byte-ok száma mindenképp 4-el osztható érték kell, hogy legyen. Ennek első sorban az az oka, hogy az alkalmazás jövőbeli verziói más processzorokkal is kompatibilisek lehetnek, és nincs rá garancia, hogy ezek közül mindegyik lehetőséget biztosít a memóriához byte-onként való hozzáférést JTAG-en keresztül (a Cortex-M3 például tud ilyet). Ezt követően lehetőséget ad újabb mérési adat felvételére, vagy a mérési adatok megadásának abbahagyására.



3.7. ábra

Mérési adatok bekérése a felhasználótól

A program minden megadott mérés után ellenőrzi, hogy az összesen kiolvasandó byte-ok száma nem haladja-e meg a maximumot (amennyiben ez megtörténik az utolsó megadott mérést elveti, és felajánlja, hogy a felhasználó újra megadja). Ha a felhasználó nem akar több mérést definiálni, megadhatja a mérések számát. A mérések száma, és az interruptok közti idő alapvetően meghatározza a mérés időtartamát, így fontos, hogy a felhasználónak lehetősége legyen beállítani. Lehetséges továbbfejlesztési lehetőség, hogy a program felajánlja, hogy annyi mérést végezzen, amennyit csak tud (ezt a rendelkezésre álló memória mérete korlátozza), így ha ez a felhasználó szándéka, akkor nem kell manuálisan kiszámolnia, hogy ez hány mérést jelent pontosan. Miután a program sikeresen beolvasta a mérési paramétereket, elhelyezi őket a memóriában, a számukra dedikált helyekre, amelyek ezek mind a fenntartott memóriaterületen helyezkednek el. Ahhoz, hogy az interrupt rutin sikeresen el tudja végezni a mérést, a következő adatokra van szüksége:

- Az eredeti interrupt rutin címe
- A mérési táblázat hossza
- A mérési táblázat
- A mérések száma

Ezeken kívül az is elengedhetetlen, hogy a PC-s alkalmazás beállítsa a mérési indikátorát, jelezve a mikorokontrollernek, hogy a mérés kezdetét veheti.

3.6 Mért adatok kiolvasása és kiértékelése

A PC-s alkalmazás azonnal elkezdi a mért adatok olvasását, amint beállította a mérési indikátorát. Az IT rutin ellátja az egyes méréseket egy azonosítóval, ami nem más, mint egy 32 bites számláló értéke, amely minden mérés alkalmával növekszik. Ennek egy jövőbeli felhasználási módja lehet az, hogy az adatok naplózása során figyelembe vesszük, hogy az előző méréshez képest változott-e egy bizonyos adat, és ha nem, azt nem naplózzuk. Ezt az azonosítót minden mérés elejére elhelyezi az IT rutin. A PC-s alkalmazás minden kiolvasott mérés után ellenőrzi, hogy a mérés még tart-e. Ez úgy történik, hogy a mikrokontroller indikátorát a PC-s alkalmazás olvassa, és amennyiben már nem a meghatározott bitmintát tartalmazza, azt úgy veszi, hogy a mérésnek vége. Ha úgy érzékeli, hogy vége a mérésnek, azt jelzi a felhasználónak. Ez nem ad pontos adatot arról, hogy meddig tartott a mérés (hiszen egy mérés kiolvasása

másodpercekig is tarthat), de legalább tájékoztatja a felhasználót, hogy a mérés már befejeződött, és már csak a kiolvasás van vissza. A mért adatokat az alkalmazás nem tárolja el a programon belül, hanem rögtön kiolvasás után naplózza őket egy .csv fájlba táblázatos formában. A kiolvasás és naplózás állását egy %-os érték megjelenítésével jelzi a program, így a felhasználó folyamatos visszajelzést kap a programtól Így elkerülhető az olyan eset, amikor egy hosszas művelet látszólagos tétlensége arra enged következtetni, hogy a program lefagyott.

A kiértékelésben egy excel táblázat segít, amiben egy adatcsatlakozás van a mérési adatokat tároló .csv fájlhoz. Ezt a csatlakozást frissítve betöltődnek a legfrissebb mért adatok a kiértékelő táblázatba. Ekkor még mindig byte-okra van bontva az az eredmény, így a táblázat feladata, hogy összeállítsa belőlük az egyes változókat. Ha megvannak az egyes sorok „feloldásai” (pl. egy 4 byte-ból álló sor feloldása lehet 4 db 1 byte-os változó értéke, vagy 1 db 4 byte-os változó értéke), akkor az így kapott változóértékeket érdemes külön oszlopba kimásolni, úgy, hogy egy oszlop egyetlen változó időbeli változását mutassa. Ezekből az oszlopokból már könnyen készíthető grafikon, és könnyebben exportálható eredményt is ad.

4 Tesztelés

A tesztek során, az eszközön futó programban egy végtelen ciklus futott, amiben bizonyos feltételek mellett növekedtek változók. A mérésre használt interrupt a SysTick interrupt volt, ami 20ms-os periódusidőre lett inicializálva. A SysTick eredeti funkciója a programban, hogy minden lefutásakor megváltoztassa egy, a panelen lévő led állapotát (ha ég, kikapcsolja, ha nem ég, bekapcsolja). Ez lehetőséget biztosít egyrészt arra, hogy ellenőrizzük valóban lefut-e az eredeti interrupt függvény (villog a led), másrészt, hogy nem befolyásolja-e túlzottan a led villogási periódusidejét a mérés. Az egyik ilyen teszten négy változó került mérésre:

- „A” változó (32 bites egész): minden iterációban 1-el nő (kezdőértéke 0)
- „B” változó (32 bites egész): minden iterációban 1-el csökken (kezdőértéke 2^{31})
- „C” változó (32 bites egész): minden 10000 iteráció után 1-el nő (kezdőértéke 0)
- „D” változó (4 db 32 bites elemből álló tömb): egyik eleme egyesével nő minden 5000 iteráció után, másik eleme kettesével stb.

Ebből a tesztből kiderült, hogy noha egy tömb elemeit is képes korrektül mérni az alkalmazás, ezeknek megjelenítése a naplófájlban nem éppen átlátható. Amennyiben úgy adjuk meg a mérési paramétereket, hogy a tömb elejéről x byte-ot akarunk olvasni (ahol x a tömb teljes hossza byteokban), akkor a kapott táblázatban egy sort fog elfoglalni a tömb, és az elemei közt nehéz ránézésre határokat húzni. Ez teljesen megfelel a várt működésnek, de gyakorlati szempontból hátrányos. Egy jövőbeli fejlesztése lehet a programnak, hogy lehetőséget biztosít kifejezetten tömbök (vagy egyéb adatstruktúrák) nyomon követésére, amik egyszerű változóktól eltérő módon jelennek meg a naplófájlban.

5 Értékelés

A szakdolgozat témája egy mérőalkalmazás megtervezése és megvalósítása, amely JTAG kapcsolaton keresztül képes valós idejű méréseket végezni egy beágyazott rendszerben, anélkül, hogy jelentősen zavarná az eredeti program működését.

Az elkészült mérőalkalmazás mindkét szoftverkomponense (az IT rutin, és a PC-s alkalmazás) a kívántaknak megfelelően működik, és a tesztek során a várt eredményeket hozták.

Kétségtelen, hogy az alkalmazás számos területen fejlődhet még, és a fejlesztés során igyekeztem biztosítani ezeknek a továbbfejlesztéseknek az akadálymentességét.

5.1 Továbbfejlesztési lehetőségek

5.1.1 Sebességnövelés

A megvalósított alkalmazás képes elvégezni a kiszabott feladatot, de a mért adatok kiolvasása időigényes lehet (főleg ha sok adat tartozik egy méréshez, vagy ha sok mérésről van szó). Az egyik terület ahol ez javítható lenne, az az AP és DP regiszterekhez való hozzáférés. Mint ahogy a 2.2 fejezet is írja, megvalósítható az adatregiszterek párhuzamos írása és olvasása (a regiszter egyik oldalán beléptetés, a másik oldalán kiléptetés). Erre lehetőséget is biztosítanak bizonyos MPSSE parancsok, így gyorsítani lehetne a DP és AP regiszterek írását, ha egyszerre olvasnánk belőlük az előző hozzáférés sikerességét (ACK kód), és írnánk beléjük a kívánt értéket. Ez egy fajta pipeline szervezése lenne az írásnak, ami lényegesen hatékonyabb, mint a jelenlegi. Amennyiben kellő mértékben sikerül a memória hozzáférés gyorsaságát növelni, a program módosítható úgy, hogy a 3.4.1. alfejezetben ismertetett RoundFIFO alapú tárolást alkalmazza a sorfolytonos helyett, amelyet a 3.4.2 alfejezet tárgyal.

Ide tartozhat még az is, hogy ha esetlegesen olyan finom időzítésű programmal kell használni a mérőalkalmazást, amit már hátráltat az injektált IT rutin lefutásának megvárása, akkor annak a kódját is lehet sebességre optimalizálni. Jelenleg (a kód olvashatósága érdekében) az IT rutinon belüli memóriaelérések legtöbbször saját pointer változón keresztül történnek. Ezeknek száma csökkenthető, ha „újrahasznosítjuk” őket. Használható pl. ugyanaz a pointer a mikrokontroller mérési

indikátorának beállítására, mint amivel a PC mérési indikátorát kiolvassuk, persze ezzel nehéz lenne beszédes változónevet adni a pointernek, amivel csökkenne a kód átláthatósága.

5.1.2 Grafikus felhasználói felület

A program jelenlegi formájában még „elfér” egy konzolos alkalmazás keretein belül, de további komplexitás hozzáadásával hamar átláthatatlanná válna. A jövőben célszerű volna átemelni a felhasználói kezelőfelületet egy ablakos nézetbe, ahol a mérés paraméterei táblázatban, vagy szövegdobozokban adhatóak meg. Byte számok helyett az is megadható, hogy milyen változót akarunk mérni (char, unsigned int, long stb.), és hogy milyen körülmények között (milyen nyelv, hány bites processzor, milyen compiler). Ezekből az adatokból a program magától meg tudná határozni, hogy hány byte-ot kell mérnie.

5.1.3 Kiterjesztés más processzorokra

A megvalósított alkalmazás egy Cortex-M3 alapú LPC1769 processzorra készült, de minimális változtatásokkal más Cortex processzorokra is alkalmazható lenne. Az elsődleges akadálya a más processzorokkal való használatnak, hogy néhány lépés architekturális sajátosságokra alapoz (mint pl. a DHCSR regiszter, az interrupt vektor áthelyezhetősége a RAM-ba, vagy a CoreSight debug port), ami más processzoroknál teljesen eltérő lehet. A program fejlesztése során külön egységekbe lettek szervezve azok a kódrészek, amik univerzálisan alkalmazhatóak bármely JTAG kompatibilis eszközre, és azok is, amik az ARM Debug Interface-re alapoznak. Olyan adatok is külön lettek szedve, amelyek architektúrától vagy processzortól függhetnek (mint pl. egy-egy JTAG utasítás kódja), így ezek módosításához nem szükséges a kódot részletesen ismerni. Ahhoz hogy a felhasználó a processzor típusát is kényelmesen be tudja állítani, már elengedhetetlen a 5.1.2 alfejezetben említett grafikus felület.

5.1.4 Fizikai kapcsolat véglegesítése

A szakdolgozat elején tárgyalt kapcsolódási problémát ugyan megoldja az átalakított szalagkábel, de a felbontott erek rendkívül vékonyak és sérülékenyek. Tartós mechanikai igénybevétel hatására könnyen elképzelhető, hogy az erek elszakadnak, vagy a forrasztások elengednek. A jövőben célszerű volna egy alkalmas csatlakozó készítése, ami nem igényli a szalagkábel felbontását.

Irodalomjegyzék

[1] *UM10360 LPC176x/5x Rev. 4.1. – 19 December 2016*
http://www.nxp.com/documents/user_manual/UM10360.pdf

[2] *C232HM USB 2.0 Hi-Speed to MPSSE Cable Datasheet v1.2.1 2016-05-05*
http://www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_C232HM_MPSSE_CABLE.PDF

[3] *Cortex-M Debug Connectors*
http://infocenter.arm.com/help/topic/com.arm.doc.faqs/attached/13634/cortex_debug_connectors.pdf

[4] *IEEE 1149.1 Standard for Test Access Port and Boundary-Scan Architecture (2013)*

[5] *Application Note AN_108 Command Processor for MPSSE and MCU Host Bus Emulation Modes v1.5*
http://www.ftdichip.com/Support/Documents/AppNotes/AN_108_Command_Processor_for_MPSSE_and_MCU_Host_Bus_Emulation_Modes.pdf

[6] *Application Note AN_129 Interfacing FTDI Hi-Speed USB Devices to a JTAG TAP v1.1*
http://www.ftdichip.com/Support/Documents/AppNotes/AN_129_FTDI_Hi_Speed_USB_To_JTAG_Example.pdf

[7] *Efficient Control of JTAG TAP*
http://www.nxp.com/assets/documents/data/en/supporting-information/BeyondBits2article05.pdf

[8] *ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*

[9] *ARM infocenter – NVIC register descriptions*
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337e/Cihcbadd.html

[10] *ARM infocenter – Debug Halting Control and Status Register*
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337e/CEGCJAHJ.html

Függelék

A függelék tartalmazza

- a PC-s alkalmazás main() függvényét
- a mérést elvégző IT rutint
- egy példamérés kimenetét, és annak kiértékelését

PC-s alkalmazás main() függvénye:

```
int _tmain(int argc, _TCHAR* argv[])
{
	// Handle of the FTDI device
	FT_HANDLE ftHandle;
	// The conn.ftStatus will point to this variable
	FT_STATUS ftStatus = FT_OK;
	// Value of clock divisor, SCL Frequency = 12/((1+dwClockDivisor)*2) (MHz)
	DWORD dwClockDivisor = 5;
	// Current state of the TAP Controller state machine
	TAP_STATE state = TAP_TLR;
	// Struct containing variables related to the JTAG connection
	jtagConnection conn;
	// FTDI handle
	conn.ftHandlePtr = &ftHandle;
	// Result of each D2XX call
	conn.ftStatusPtr = &ftStatus;
	// JTAG TAP state machine state
	conn.statePtr = &state;
	// Starting address of the "new" IT vector
	long itVectorAddress;
	// Table containing the measurement parameters (filled up when the user
provides the input)
	measurementParameters measurementTable[MAX_BYTECOUNT];
	// Length of the measurement table
	unsigned int tableLength = 0;
	// Error code
	int errorStatus = ERR_OK;

	// Initializing FTDI device
	errorStatus |= initDevice(conn);
	// Configuring FTDI device for MPSSE commands
	errorStatus |= configDevice(conn);
	// Synchronizing with FTDI device
	errorStatus |= syncDevice(conn);
	// Configuring parameters of JTAG transfer
	errorStatus |= configJTAG(conn, dwClockDivisor);
	// Checking ID of the JTAG TAP
	errorStatus |= checkDeviceId(conn);

	if (errorStatus != ERR_OK) return 1;
	// Resetting the TAP controller
	*conn.statePtr = navigateToTestLogicReset(conn);

	// Selecting the first access port in the line (which we know is a MEM-AP)
	errorStatus |= selectAccessPort(conn, 0x00000000);
```

```
	// Clearing sticky error flags from the CTRL/STAT Debug Port register
	clearStickyErrorFlags(conn);
	// Requesting powerup from the System and Debug port
	errorStatus |= requestSysDbgPowerup(conn);

	// Setting up new address for interrupt vector
	itVectorAddress = getValidVtorAddr(NVIC_SPACE_ADDR);
	// Remapping the IT vector to the determined address
	errorStatus |= remapItVector(conn, itVectorAddress);
	// Switching the interrupt address with the address of the newly loaded IT
routine (+1 at the address is due to a constraint of the Cortex-M3 processor)
	errorStatus |= switchInterrupt(conn, (itVectorAddress + NVIC_OFFSET_SYSTICK),
IT_ROUTINE_ADDR);
	// Erasing log file
	errorStatus |= flushLog(LOGFILE_PATH);
	// Getting measurement parameters from user
	errorStatus |= promptParameters(measurementTable, &tableLength,
&numMeasurements);
	// Initializing measurement (placing parameters at their pre-defined places
in the memory
	errorStatus |= initMeasurementParameters(conn, measurementTable, tableLength,
numMeasurements);
	// Performing measurement
	errorStatus |= startMeasurement(conn, measurementTable, tableLength,
numMeasurements);


	// Checking for any errors during DP/AP accesses (this function will print to
the console, if there were any)
	if(getStickyErrorFlags(conn)) errorStatus |= ERR_RW;
	//Closing everything down
	errorStatus |= goToState(conn, TAP_RTI);	// Putting TAP controller in
idle state
	if ((errorStatus == ERR_OK) && FT_SUCCESS(*conn.ftStatusPtr) &&
*conn.statePtr != ALL_TAP_STATES) printf("\nAll commands executed successfully\n");
	else printf("\nAn error has occurred while executing the commands\n");


	FT_Close(ftHandle);				// Close the port
	return 0;					// Exit with success
}
```

IT rutin:

```
int main(void) {
	typedef void func(void);

	typedef struct {
		uint32_t address;
		uint32_t byteCnt;
	} measurementParameters;
	measurementParameters mParam[MAX_BYTECOUNT];  /*If each entry has byteCnt =
1, we can store a maximum of MAX_BYTECOUNT table entries*/

	/*Some pointer pairs are only used in certain sections of the program. To
conserve memory, these pairs are implemented as union members. Two members of the
same union are never used together, and are never dependent on each other.*/
	union pinters0 {
		uint32_t* pcMeasurementIndicator;
		uint8_t* readPtr;
		uint32_t* addressPtr;
	} ptrs0;

	union pointers1 {
		uint32_t* ucMeasurementIndicator;
		uint8_t* writePtr;
		uint32_t* byteCntPtr;
	} ptrs1;

	union ptrCast  {
		func**		pp2Funct;
		func*		p2Funct;
		uint32_t*   addressPtr;
	} fptrCast;


	/*Calling the original function, the address of which is stored at
gd_ORIGINAL_IT_ROUTINE_ADDR_ADDR_ui32*/
	fptrCast.addressPtr = ORIGINAL_IT_ROUTINE_ADDR_ADDR;
	func** f = fptrCast.pp2Funct;
	(*f)();

	uint32_t* completedMeasurementsPtr = COMPLETED_MEASUREMENTS_ADDR;
	uint32_t* numMeasurementsPtr = NUM_MEASUREMENTS_ADDR;
	ptrs1.ucMeasurementIndicator = UC_MEASUREMENT_IND_ADDR;

	/*If the amount of completed measurements is over the number of desired
measurements, we return here.*/
	if ((*completedMeasurementsPtr + 1) > (*numMeasurementsPtr))
	{
		*ptrs1.ucMeasurementIndicator = 0x00000000;	/*Clearing
measurement indicator*/
		return 0;
	}

	if (*ptrs1.ucMeasurementIndicator != INDICATOR_PATTERN)
*ptrs1.ucMeasurementIndicator = INDICATOR_PATTERN;  /*Setting the uC measurement
indicator (this will let the PC know, that the polling can begin)*/

	ptrs0.pcMeasurementIndicator = PC_MEASUREMENT_IND_ADDR;
	if (*ptrs0.pcMeasurementIndicator != INDICATOR_PATTERN) return 0;
                                                    /*If the PC has not yet
set its measurement indicator, the measurement needs not take place*/

	ptrs0.addressPtr = MEASUREMENT_TABLE_ADDR;
```

```
	ptrs1.byteCntPtr = MEASUREMENT_TABLE_ADDR + 4;
	uint8_t totalBytes = 4;	/*This accounts for the 4 bytes at the start of
each measurement entry, that store the current value of the completed measurements
counter*/

	uint32_t* numMeasurementTableEntries = MEASUREMENT_TABLE_ENTRIES_ADDR;
	/*Reading measurement parameters from the measurement table*/
	for (int i = 0; i < (*numMeasurementTableEntries); i++)
	{
		mParam[i].address = *ptrs0.addressPtr;
		mParam[i].byteCnt = *ptrs1.byteCntPtr;
		if ((*ptrs1.byteCntPtr) <= MAX_BYTECOUNT) totalBytes +=
*ptrs1.byteCntPtr;
		else break;
		ptrs0.addressPtr = ptrs0.addressPtr + 2;
		ptrs1.byteCntPtr = ptrs1.byteCntPtr + 2;
	}

	ptrs1.writePtr = MEASURED_DATA_ADDR + (totalBytes *
(*completedMeasurementsPtr));		/*This will set the writePtr to the start
of the current measurement entry*/

	if ((ptrs1.writePtr + totalBytes) > (MEASURED_DATA_ADDR +
MEASURED_DATA_MAX_SIZE))
	{
		ptrs1.ucMeasurementIndicator = UC_MEASUREMENT_IND_ADDR;
		*ptrs1.ucMeasurementIndicator = 0x00000000;	// Clearing
measurement indicator
		return 0;
	}

	if (*completedMeasurementsPtr == 1)
	{
	uint32_t* logPtr = LOG_FIELD_ADDR;
	*logPtr = ptrs1.writePtr;
	logPtr++;
	*logPtr = (*completedMeasurementsPtr + 1) & 0xFF;
	logPtr++;
	*logPtr = ((*completedMeasurementsPtr + 1) & 0xFF00) >> 8;
	logPtr++;
	*logPtr = ((*completedMeasurementsPtr + 1) & 0xFF0000) >> 16;
	logPtr++;
	*logPtr = ((*completedMeasurementsPtr + 1) & 0xFF000000) >> 24;
	logPtr++;
	}
	/*This measurement entry belongs to the (*completedMeasurementsPtr + 1)th
measurement (the completedMeasurements counter is only incremented at the end of
each measurement)*/

	*ptrs1.writePtr = (*completedMeasurementsPtr + 1) & 0xFF;
	ptrs1.writePtr++;
	*ptrs1.writePtr = ((*completedMeasurementsPtr + 1) & 0xFF00) >> 8;
	ptrs1.writePtr++;
	*ptrs1.writePtr = ((*completedMeasurementsPtr + 1) & 0xFF0000) >> 16;
	ptrs1.writePtr++;
	*ptrs1.writePtr = ((*completedMeasurementsPtr + 1) & 0xFF000000) >> 24;
	ptrs1.writePtr++;// Setting the writePtr to the start of the data field of
this entry

	/*Performing measurement*/
	for (int i = 0; i < (*numMeasurementTableEntries); i++)
	{
		ptrs0.readPtr = mParam[i].address;
		for (int j = 0; j < mParam[i].byteCnt; j++)
		{
			*ptrs1.writePtr = *ptrs0.readPtr;
```

```
                ptrs1.writePtr++;
                ptrs0.readPtr++;
            }
        }

        (*completedMeasurementsPtr)++;
        return 0;
}
```

Példamérés:

A példamérés során két 4 byte-os változó értéke került mérésre, az egyik (0x10002014) egy while(1) ciklus minden lefutásakor inkrementálódott, a másik (0x10002018) minden 10000. Lefutás alkalmával.

Measurement ID	Address	Bytes									
1	0x10002014	3	a5	5a	3d	03a55a3d	61168189		Number of measurements		
1	0x10002018	0	0	17	e4	000017e4	6116		100		
2	0x10002014	3	a5	62	9e	03a5629e	61170334		Number of measured variables		
2	0x10002018	0	0	17	e5	000017e5	6117		2		
3	0x10002014	3	a5	6a	ff	03a56aff	61172479		Time interval between interrupts (in ms)		
3	0x10002018	0	0	17	e5	000017e5	6117		20		
4	0x10002014	3	a5	73	61	03a57361	61174625				
4	0x10002018	0	0	17	e5	000017e5	6117				
5	0x10002014	3	a5	7b	c2	03a57bc2	61176770				
5	0x10002018	0	0	17	e5	000017e5	6117				
6	0x10002014	3	a5	84	24	03a58424	61178916				
6	0x10002018	0	0	17	e5	000017e5	6117				
7	0x10002014	3	a5	8c	85	03a58c85	61181061				
7	0x10002018	0	0	17	e6	000017e6	6118				
8	0x10002014	3	a5	94	e7	03a594e7	61183207				
8	0x10002018	0	0	17	e6	000017e6	6118				
9	0x10002014	3	a5	9d	48	03a59d48	61185352				
9	0x10002018	0	0	17	e6	000017e6	6118				
10	0x10002014	3	a5	a5	aa	03a5a5aa	61187498				
10	0x10002018	0	0	17	e6	000017e6	6118				
11	0x10002014	3	a5	ae	c	03a5ae c	61189644				
11	0x10002018	0	0	17	e6	000017e6	6118				
12	0x10002014	3	a5	b6	6d	03a5b66d	61191789				
12	0x10002018	0	0	17	e7	000017e7	6119				
13	0x10002014	3	a5	be	cf	03a5becf	61193935				
13	0x10002018	0	0	17	e7	000017e7	6119				
14	0x10002014	3	a5	c7	30	03a5c730	61196080				
14	0x10002018	0	0	17	e7	000017e7	6119				
15	0x10002014	3	a5	cf	92	03a5cf92	61198226				
15	0x10002018	0	0	17	e7	000017e7	6119				
16	0x10002014	3	a5	d7	f3	03a5d7f3	61200371				
16	0x10002018	0	0	17	e8	000017e8	6120				
17	0x10002014	3	a5	e0	54	03a5e054	61202516				
17	0x10002018	0	0	17	e8	000017e8	6120				
18	0x10002014	3	a5	e8	b6	03a5e8b6	61204662				
18	0x10002018	0	0	17	e8	000017e8	6120				
19	0x10002014	3	a5	f1	17	03a5f117	61206807				
19	0x10002018	0	0	17	e8	000017e8	6120				
20	0x10002014	3	a5	f9	79	03a5f979	61208953				
20	0x10002018	0	0	17	e8	000017e8	6120				
21	0x10002014	3	a6	1	da	03a601da	61211098				
21	0x10002018	0	0	17	e9	000017e9	6121				
22	0x10002014	3	a6	a	3c	03a6 a3c	61213244				
22	0x10002018	0	0	17	e9	000017e9	6121				
23	0x10002014	3	a6	12	9d	03a6129d	61215389				

evaluation | variable1 | variable2 | variable3 | variable4 | variable5 | variable6

5.1. ábra

Példamérés táblázat 4 byte-os változókra felállítva

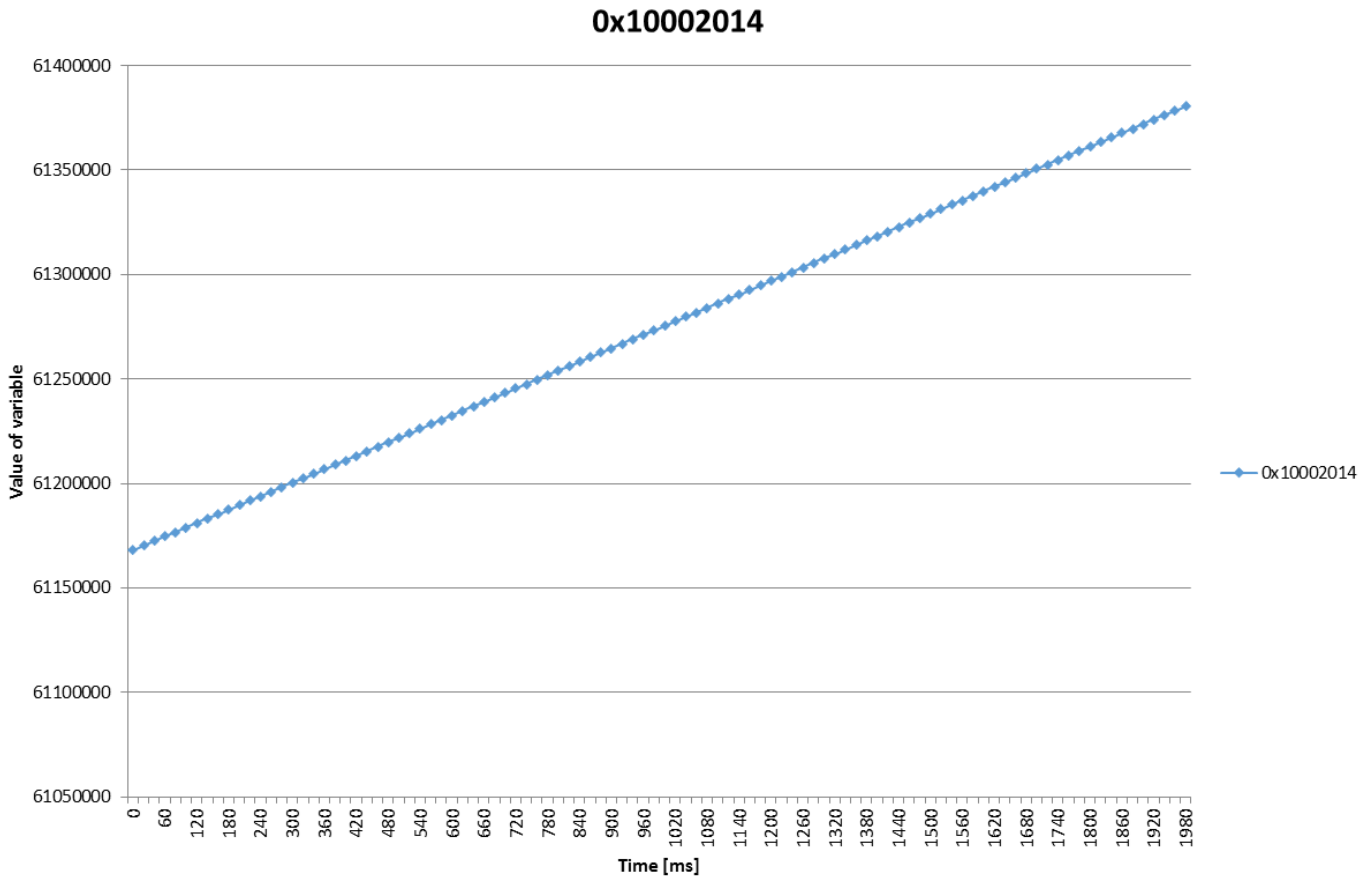
Az 5.1. ábra táblázatának első oszlopa az egyes mérések azonosítója, a második oszlop pedig a vizsgált cím. Értelemszerűen egy méréshez annyi sor tartozik, ahány cím volt vizsgálva. Az ezt követő négy oszlopban a címekről mért byte-ok találhatóak (egyesével, hexadecimális formában), majd az ötödik oszlopban a négy byte összefűzve is szerepel. A hatodik oszlop már decimális formában szerepelteti az adott sorhoz tartozó adatot.

N	O	P	Q	R	S	T
	variables					
Time (in ms)	0x10002014	0x10002018				
0	61168189	6116	-1	-1	-1	-1
20	61170334	6117	-1	-1	-1	-1
40	61172479	6117	-1	-1	-1	-1
60	61174625	6117	-1	-1	-1	-1
80	61176770	6117	-1	-1	-1	-1
100	61178916	6117	-1	-1	-1	-1
120	61181061	6118	-1	-1	-1	-1
140	61183207	6118	-1	-1	-1	-1
160	61185352	6118	-1	-1	-1	-1
180	61187498	6118	-1	-1	-1	-1
200	61189644	6118	-1	-1	-1	-1
220	61191789	6119	-1	-1	-1	-1
240	61193935	6119	-1	-1	-1	-1
260	61196080	6119	-1	-1	-1	-1
280	61198226	6119	-1	-1	-1	-1
300	61200371	6120	-1	-1	-1	-1
320	61202516	6120	-1	-1	-1	-1
340	61204662	6120	-1	-1	-1	-1
360	61206807	6120	-1	-1	-1	-1
380	61208953	6120	-1	-1	-1	-1
400	61211098	6121	-1	-1	-1	-1
420	61213244	6121	-1	-1	-1	-1
440	61215389	6121	-1	-1	-1	-1
460	61217535	6121	-1	-1	-1	-1
480	61219681	6121	-1	-1	-1	-1
500	61221826	6122	-1	-1	-1	-1
520	61223972	6122	-1	-1	-1	-1
540	61226117	6122	-1	-1	-1	-1
560	61228263	6122	-1	-1	-1	-1
580	61230408	6123	-1	-1	-1	-1
600	61232554	6123	-1	-1	-1	-1
620	61234700	6123	-1	-1	-1	-1
640	61236845	6123	-1	-1	-1	-1
660	61238990	6123	-1	-1	-1	-1
680	61241136	6124	-1	-1	-1	-1
700	61243281	6124	-1	-1	-1	-1
720	61245427	6124	-1	-1	-1	-1
740	61247572	6124	-1	-1	-1	-1
760	61249718	6124	-1	-1	-1	-1
780	61251863	6125	-1	-1	-1	-1
800	61254009	6125	-1	-1	-1	-1
820	61256154	6125	-1	-1	-1	-1
840	61258300	6125	-1	-1	-1	-1
860	61260445	6126	-1	-1	-1	-1

5.2. ábra

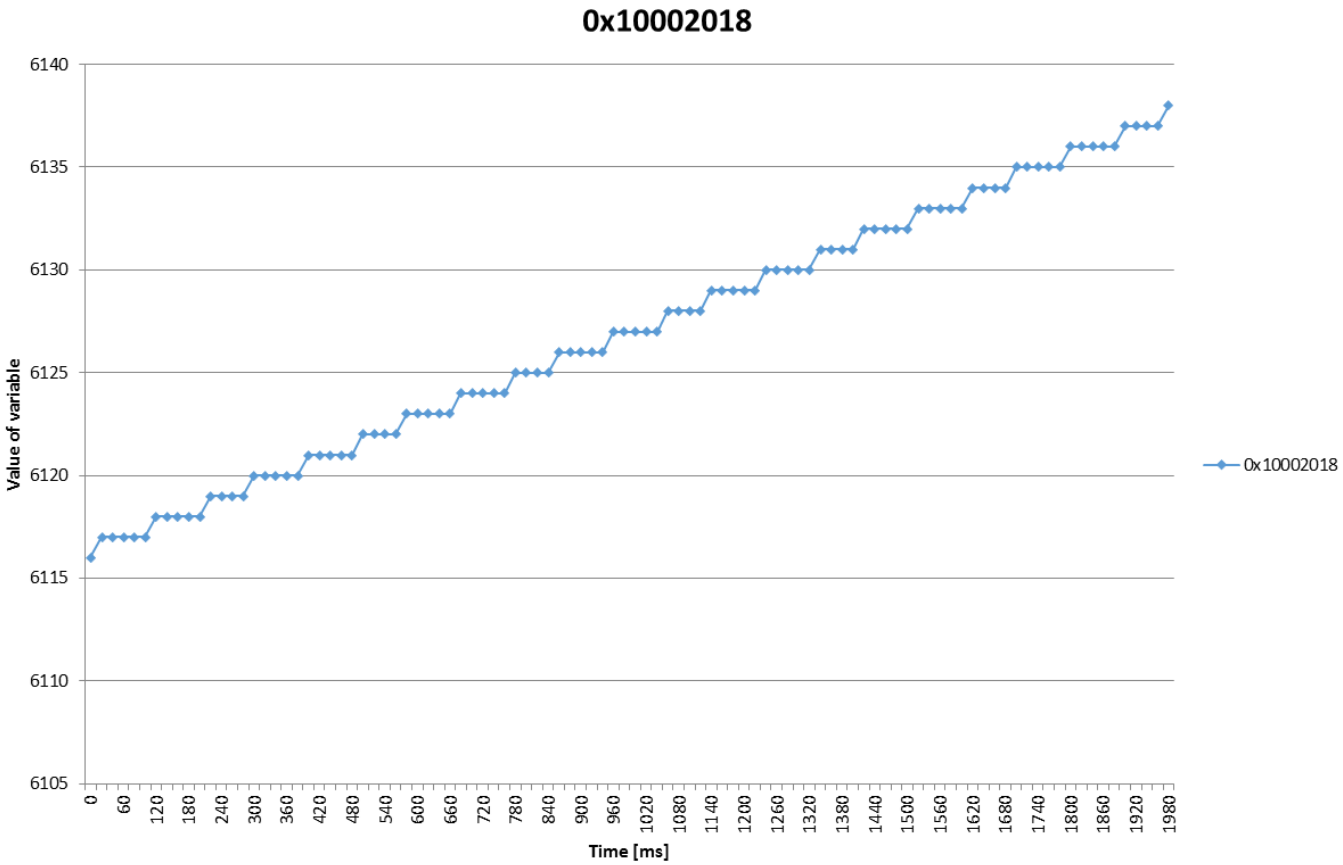
Példamérés adatai változónként oszlopokba szedve

Az 5.2. ábra már külön szedve tertalmazza az egyes mért változókat, és azok változását az időben. A táblázat ezen része automatikusan frissül, ha a felhasználó változtatja a mérés adatait (ld. 5.1. ábra jobb oldalán). A nem használt oszlopok -1-el vannak feltöltve, hogy ne okozzanak gondot az Excel képleteinek. Az így kapott oszlopokból már készíthető diagram az alábbi ábrákon látható formában.



5.3. ábra

while(1) ciklus minden lefutásakor növelt változó értéke az idő függvényében



5.4. ábra

while(1) ciklus minden 10000. Lefutásakor növelt változó az idő függvényében