Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Logsys fejlesztőeszköz Linux környezetben

TDK dolgozat

Készítette:

Csókás Bence Viktor

Konzulens:

dr. Fehér Béla
Raikovich Tamás

2019

Tartalomjegyzék

Kivonat

A TDK munkám célja a tanszéki (és kari) oktatásban használt Logsys rendszer platformtámogatottságának kibővítése.

A XILINX ISE fejlesztőkörnyezet elérhető Linux alatt, azonban a Logsys Download Cable kezelése csak Windows alatt megoldott, a Logsys GUI nevű segédprogrammal. Mivel én kizárólag Linuxot használok, a Logsys eszköz használatához egy virtuális gépet fenn kellett volna tartanom.

Munkám során ezt a segédprogramot fejtettem vissza, majd az így szerzett tapasztalataimmal egy platformfüggetlen, LibUSB alapú eszközmeghajtót hoztam létre. Az eszköz kezelésére és a meghajtóprogram tesztelésére egy parancssori példaprogramot írtam. A programot teszteltük több Linux disztribúción (Ubuntu, Arch, Mint, Fedora), macOS-en és Haiku-n is.

A példaprogram jelenleg képes a letöltőkábel automatikus detektálására, a tápellátás biztosítására, a JTAG interfészen a csatlakoztatott FPGA (vagy más programozható eszköz) detektálására és a konfigurációs kód letöltésére. Ezen felül az órajel kimenet és a RESET beállítására, az eszköz állapotleíróinak (vonalak állapota, kifolyó áramerősség, túláramvédelem küszöbértéke, aktív átviteli mód stb.) lekérdezésére. Ezenfelül a driver fel van készítve néhány további funkcióra is, amik a parancssori példaprogramba még nem kerültek beépítésre: ilyen a BitBang I/O kezelés, USART és SPI adatátvitel. Ezen funkciók tesztelésére szintén írtam kisebb példaprogramokat.

A jövőbeni továbbfejlesztés során a következő hiánypótlásokat lehetne eszközölni: Logsys GUI reprodukálása, a nem dokumentált átviteli módok (I^2C, további 2 vagy többvezetékes programozói interfészek) támogatása, az USART átvitelhez TTY (TeleTYpe, karakteres kimeneti eszköz) létrehozása, valamint a MiniRISC nevű softcore CPU hardveres debuggerének kezelése.

A projektben hivatkozom Raikovich Tamás diplomatervére, valamint Wacha Gábor és Kiss Benedek programkódjaira.

Abstract

The goal of my work is to extend the platform support of the Logsys toolchain, currently in use in education at the department and the faculty.

XILINX's ISE is already available on Linux, but use of the Logsys Download Cable requires running a Windows-only utility program, called Logsys GUI. As I use Linux exclusively, running such a program would have required that I maintain a VM.

During my work, I reverse engineered this tool and created a platform-independent, LibUSB-based driver. For using and testing of the driver, I created a command-line program, that ran successfully on a handful of Linux distros (Ubuntu, Arch, Mint, Fedora), along with macOS and Haiku.

The program is capable of recognizing automatically the download cable, provide power supply, detecting and identifying the FPGA (or other programmable device) on the JTAG chain and downloading the configuration code onto it. Above these it ensure the controlling of the RST pin, the setting of the internal square wave generator's clock frequency, getting device status (the state of the lines, output current, reverse current tolerance, active serial data transfer mode etc.). On top of that, the driver supports additional features not yet utilized by this program, such as BitBang I/O, USART, SPI. For these features I wrote smaller test snippets.

In the future, the following improvements could be made: reproducing the Logsys GUI, supporting the non-documented transfer modes (I^2C and other 2 or more wire programming protocol), adding a TTY for USART transfers, and supporting the MiniRISC softcore CPU's hardware debugger.

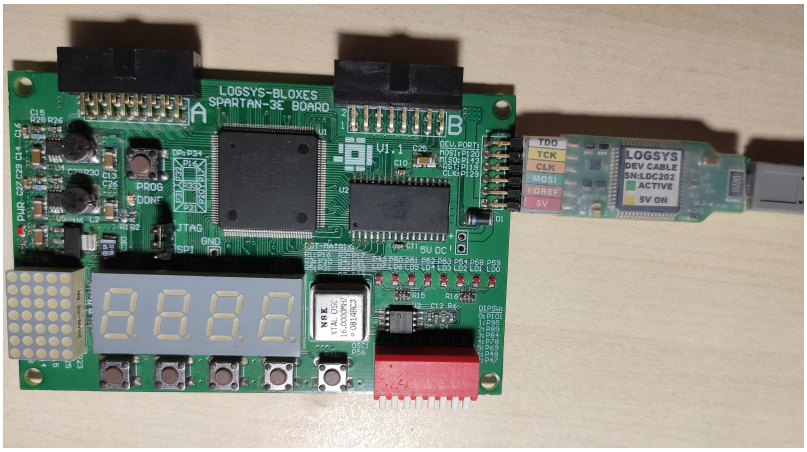
In the project I reference Tamás Raikovich's thesis draft and make use of code written by Gábor Wacha and Benedek Kiss.

1. A Logsys rendszer

A TDK munkám alapja (előzménye, ha úgy tetszik) a kari oktatásban használt, a Méréstechnika és Információs Rendszerek tanszék által kifejlesztett Logsys fejlesztői környezet. Ez a rendszer 3 részre bontható:

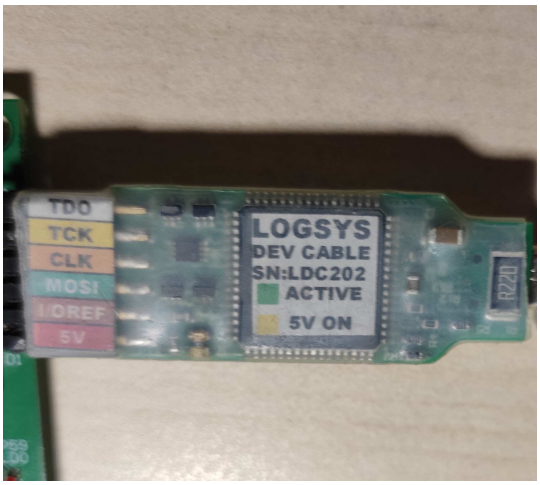
- a programozható eszközt tartalmazó kártyára (az én esetemben ez egy Logsys-BLOXES Spartan-3E Board nevű FPGA kártya),
- a letöltőkábelre (Logsys Development Cable, a továbbiakban LDC néven is fogom hivatkozni),
- illetve a számítógépen futó szoftverre.

FPGA kártya A LOGSYS Spartan-3E FPGA kártya egy egyszerű felépítésű, elsősorban kezdő felhasználók számára készült FPGA kártya [2]. A kártyán található a programozandó céleszköz, az őt támogató áramkörök (tápellátás biztosítása, kvarc oszcillátor, jumperek, a konfigurációt tároló flash stb.) valamint az általa használt perifériák.



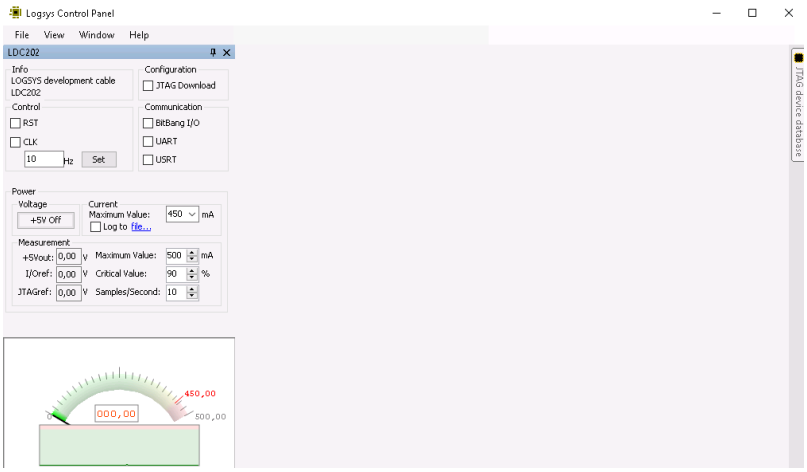
1. ábra. A Logsys-BLOXES Spartan-3E FPGA kártya, csatlakoztatva az LDC-hez

LDC A fejlesztői kábel az USB porton keresztül kapcsolja össze a célrendszert a PC-vel. Biztosít konfigurációs, vezérlési (órajel és reset jel) és soros kommunikációs interfészt és rendelkezik 5V-os tápfeszültség kimenettel is [1]. Az eszköz működéséről részletesebben a 3. fejezetben írok.



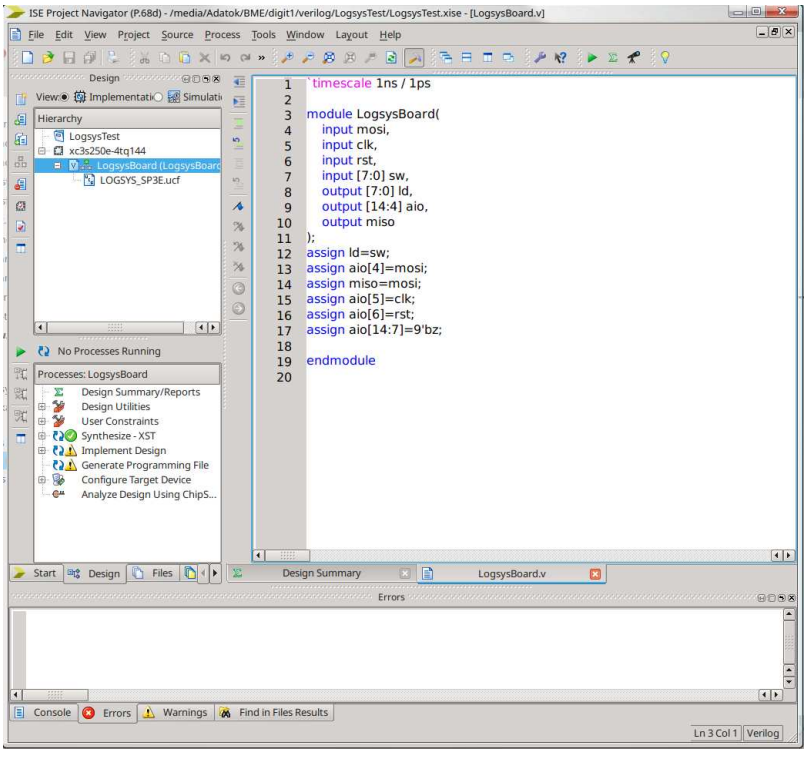
2. ábra. Az LDC202-es sorozatszámú Logsys Download Cable fejlesztői kábel

Logsys GUI Az LDC-vel való kommunikációt a Logsys GUI nevű szoftver valósítja meg. Ez a program és az eszközmeghajtók Windows operációs rendszert igényelnek [1]. A program segítségével az elkészült konfigurációt a céleszközre (jelen esetben a Spartan-3E FPGA-ra) tölthetjük, a LDC JTAG interfészét felhasználva.



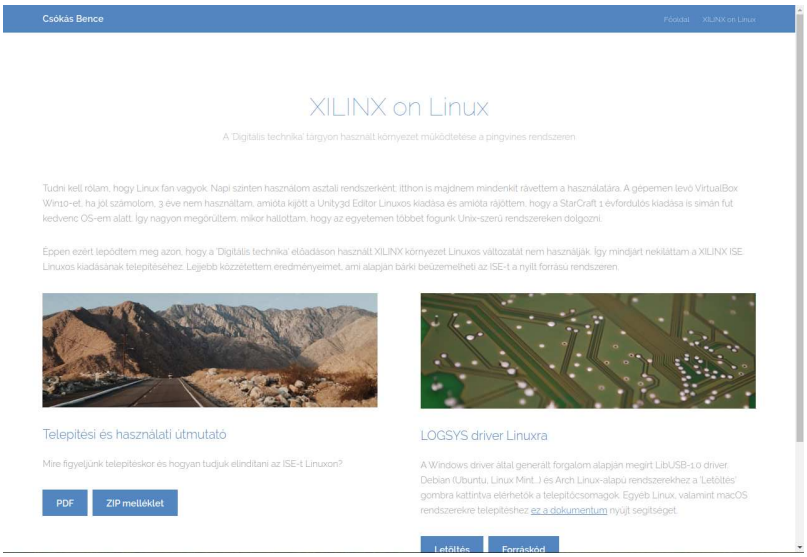
3. ábra. A Logsys GUI, miután csatlakoztattuk az LDC-t egy USB porthoz

XILINX ISE A konfigurációs fájl előállításához a XILINX ISE nevű fejlesztőkörnyezetet kellett használni, ami elérhető mind Windows, mind Linux operációs rendszereken, bár a telepítése egy kissé nehézkes. Mivel néhány hallgatótársam is érdeklődött a Linux alatti fejlesztés iránt, létrehoztam egy weboldalt (`http://users.hszk.bme.hu/~cb1719/xil-lin.php`), amin a legfrissebb eredményeimet tudták követni e téren.



4. ábra. A XILINX ISE 14.6 futása Linuxon

A XILINX ISE működésre bírása után már csak a Logsys GUI (illetve a hozzá tartozó driver) volt az egyetlen, ami nem volt elérhető Linuxon [5].

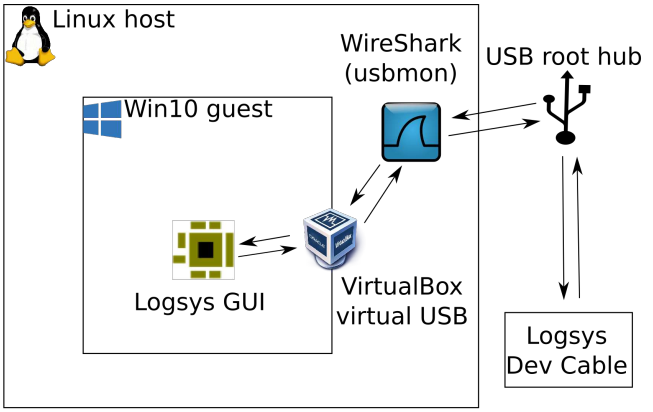


5. ábra. Az elkészült weboldal

2. Reverse engineering

Mivel a tanszék nem adta ki a forráskódokat, a platformfüggetlenítéshez először vissza kellett fejtenem az eredeti, Windowsos programot, illetve drivert. Ez nehéz feladatnak bizonyult, hiszen jobbára semmit nem tudtam a felépítéséről, azonban egy remekül sikerült összeállítással képes voltam elkezdeni.

Az ötlet a következő volt: futtassuk a Windowsos programot egy VM-ben (Windows guest), és a Linux hoston fogjuk el az USB adatforgalmat! Ez az ötlet remekül működött; a hoston WireShark [4] segítségével el tudtam kapni az USB csomagokat (USB Request Block, URB), amit ezek után kézzel fejtettem vissza.



6. ábra. A mérési elrendezés

Azonban akadt egy kis probléma: néha a VM-en belül nem lehetett megnyitni a JTAG interfészt. Ezért kénytelen voltam egy régi Windows XP-re telepíteni a Logsys toolchaint és a WireSharkot. Azonban a Windowsos WireShark nem támogatja az USB folyam lehallgatását, így erre egy újabb segédprogramot, az USBPcap-et kellett telepítenem. De végül sikerült elegendő mérési adatot összegyűjtenem, ami alapján el tudtam kezdeni a Logsys GUI és az LDC közti protokoll felderítését.

A protokoll visszafejtését kezdetben szisztematikus teszteléssel végeztem: elindítottam a WireShark (vagy az USBPcap) csomagelkapását, megnyomtam egy gombot a Logsys GUI-n, leállítottam a csomagelfogást, és kielemeztem. Ezzel a módszerrel sikeresen megfejtettem az alap funkciók (tápfeszültség állítása, aszinkron reset, adatátviteli módok be-/kikapcsolása, JTAG lánc letapogatása). Azonban bizonyos funkciókat nem lehetett ily módon megfejteni (ilyenek tipikusan az adatátvitelek, így például az egyik legfontosabb funkció, a JTAG interfészen keresztüli FPGA konfiguráció).

Így a későbbiekben nekiálltam a Logsys GUI "hagyományos" úton történő visszafejtésének is (decompile az ILSpy nevű program segítségével), amivel kiegészíthettem hiányos ismereteimet. Emellett időközben Raikovich Tamástól kaptam egy PDF-et, amiben leírja a (csomagelkapásból megfejthetetlen, bináris adatfolyamként érkező) struktúrák kiosztását. Ezenfelül fejlesztés közben jómagam is kutattam az Interneten további dokumentáció után, így akadtam például Wacha Gábor Qt alapú Logsys GUI-rekreációjára, aminek forráskódját készségesen rendelkezésemre bocsájtotta.

Ennek a három adathalmaznak (a csomagelkapások, a visszafejtett forráskód és a dokumentáció) az összevetésével teljesebb képem alakult ki az adatátviteli protokollról.

A visszafejtett protokoll leírását a 3. fejezet taglalja.

Ezzel az ismerettel neki is álltam a saját programom fejlesztésének. Az USB kommunikáció megvalósítására a LibUSB nevű, cross-platform keretrendszert alkalmaztam. Linuxos berkekben az USB kezelést javarészt ezen az API-n keresztül valósítják meg. A Windowsos segédprogramban ezt a funkciót a kernel driver látta el.

Az FPGA konfiguráció fájlok (Serial Vector Format, SVF) kezeléséhez szükségem volt még egy függvénykönyvtárra, ez a Clifford Wolf által fejlesztett libxsvf (`http://www.clifford.at/libxsvf/`). A Logsys GUI-ban ennek az analógiája az *SVF.dll*.

3. Logsys USB kommunikációs protokoll

Az LDC USB-azonosítója (Vendor ID : Product ID) *03eb:f0ff*. Az USB eszköz két interfészt tartalmaz: a 0. interfészen található egy OUT/IN endpoint-pár (0x01, 0x82) a JTAG és SPI adatok átvitelére, illetve egy másik endpoint-pár (0x03 és 0x84) az I^2C és BitBang I/O átvitelekhez. Az 1. interfészen található az USART átvitel OUT/IN endpoint-párja (0x05 és 0x86). Ennek a kiosztásnak technikai okai vannak, amiről részletesebben Raikovich Tamás diplomatervének [3] "8.2 A funkciók megvalósítása" fejezetében olvashatunk.

A legtöbb funkció gyártóspecifikus Control kódokkal vezérelhető. Az adatátvitelek az USART IN (0x86) végpontját kivéve (ami Interrupt transzfert használ) Bulk átvitelekkel történik. Lejjebb leírom az egyes funkciókat és az őket megvalósító URB-ket.

Amennyiben a kérés valamilyen struktúrát vár/ad vissza, azt egy ofszet→mező táblázattal jelzem, majd az egyes mezők leírása következik. A táblázatban a bitmezőket "(abcd efgh)" alakban reprezentálom.

A saját implementációm hivatkozásánál a fájlok nevei és a sorszámok a kód RC3 (Release Candidate 3) verziójára érvényesek!

3.1. Control kérések

3.1.1. Státusz lekérése

A Logsys GUI működése közben figyeli az LDC állapotleíróit (kimeneti feszültségek, kifolyó áram, túláram védelem bitje stb.), ezeket az információkat másodpercenként többször is lekérdezi az eszköztől.

URB Control blokk:

Művelet	*Lekérdezés*
irány	IN
bRequest	1
wValue	0
wIndex	0
hossz	12

A kérés egy struktúrát ad vissza, aminek a formátuma a következő:

ofszet	0	1	2	3	4	5	6	7	8	9	10	11
mező	U_{out}		U_{jtag}		U_{io}		$I_{out,fine}$		I_{out}		(0000 0ROV)	(0$E_2$$E_1$S $R_{st}$$C_2$$C_1$J)

- U_{out}: A kimenő feszültséget (VCC pin) figyelő ADC nyers adata
- U_{jtag}: A JTREF lábon lévő feszültséget mérő ADC adata
- U_{io}: Az IOREF csatlakozásra eső feszültség ADC adata
- $I_{out,fine}$, I_{out}: A kifolyó áramerősséget vizsgáló két (finom és durva felbontású) ADC mért adatai
- V: A tápellátást biztosító láb (5V jelű) logikai szintje
- O: Túláramvédelem státuszbitje
- R: Visszáram védelem állapotbitje
- J: JTAG pinek (TDI, TDO, TCK, TMS) foglaltsága
- C_1: CLK1 órajelgenerátor kihasználtsága
- C_2: CLK1 négyszögjelgenerátor használtsága
- R_{st}: RST láb foglaltsága
- S: Soros I/O vonalak használtsága

- E_1, E_2: A 0. interfész két endpoint-párjának foglaltsága

Implementáció

- Státusz lekérdezése: `logsys_tx_get_status` függvény (*src/shared/control.c:7*)
- Struktúra kiosztása: `struct LogsysStatus` (*include/logsys/common.h:16*)
- Struktúra feldolgozása: *include/logsys/status.h* által definiált interfész (implementáció a *src/shared/status.c* fájlban)

3.1.2. Tápfeszültség vezérlése

A tápfeszültséget lehet be- illetve kikapcsolni, a túláram- és visszáram védelmi rendszer küszöbeit lekérdezni és átállítani. Ezek a funkciók mind a bRequest=2 kéréstípusba tartoznak.

Tápfeszültség állapota:
URB Control blokk:

Művelet	*Lekérdezés*	*Beállítás*
irány	IN	OUT
bRequest	2	2
wValue	0	kívánt állapot
wIndex	0	0
hossz	1	0

Az IN kérés egy egybájtos logikai értéket (bool) ad vissza.

Implementáció

- Lekérdezés: `logsys_tx_get_vcc` (*include/logsys/control.h:35*, implementáció: *src/shared/control.c:50*)
- Állítás: `logsys_tx_set_vcc` (*include/logsys/control.h:35*, Implementáció: *src/shared/control.c:50*)

Túláramvédelem:
URB Control blokk:

Művelet	*Lekérdezés*	*Beállítás 450 mA-ra*	*Beállítás 700 mA-ra*	*Beállítás 950 mA-ra*
irány	IN	OUT	OUT	OUT
bRequest	2	2	2	2
wValue	0x0100	0x0100	0x0101	0x0102
wIndex	0	0	0	0
hossz	1	0	0	0

Az IN kérés egy 0-2 közötti számot ad vissza, ami a 450, 700 illetve 950 mA-es küszöböt jelenti. Ezeket egy enumerációba gyűjtöttem a kezelhetőség javításának érdekében.

Implementáció

- Lekérdezés: `logsys_tx_get_pwr_limit` (*include/logsys/control.h:22*, implementáció: *src/shared/control.c:31*)
- Állítás: `logsys_tx_set_pwr_limit` (*include/logsys/control.h:24*, Implementáció: *src/shared/control.c:38*)
- Enumeráció: `enum LogsysPwrLimit` (*include/logsys/common.h:30*)

Áramerősség-mérés korrekciós faktorai:
URB Control blokk:

Művelet	*Lekérdezés*
irány	IN
bRequest	2
wValue	0x0200
wIndex	0
hossz	21

Az IN kérés által visszaadott struktúrát nem implementáltam, de a következőképpen néz ki:

ofszet	0	1	2	3	4	5	6	7	8	9	10	11	12	...
mező	H	$c_{U_{out}}$				$c_{U_{jtag}}$				$c_{U_{io}}$				...

ofszet	13	14	15	16	17	18	19	20
mező	$c_{I_{out,fine}}$				$c_{I_{out}}$			

- H: hibajelző bájt, ha 0, akkor a kérés sikeres volt
- $c_{...}$: az alsó indexben feltüntetett mennyiséghez tartozó szorzótényező

Implementáció

- Lekérdezés: `logsys_tx_get_pwr_corr` (*include/logsys/control.h:30*, implementáció: *src/shared/control.c:42*)
- Struktúra: nincs implementálva

Visszáram-tolerancia
URB Control blokk:

Művelet	*Lekérdezés*	*Beállítás*
irány	IN	OUT
bRequest	2	2
wValue	0x0300	0x0300
wIndex	0	új érték
hossz	2	0

Az IN kérés egy 10 bites, fixpontos kettes komplemens számot ad vissza, az OUT kérés ugyanebben a formátumban várja az adatot. A kódom elvégzi ezek és a lebegőpontos számábrázolás közti konverziót.
Implementáció

- Lekérdezés: `logsys_tx_get_rev_curr` (*include/logsys/control.h:38*, implementáció: *src/shared/control.c:54*)
- Állítás: `logsys_tx_set_rev_curr` (*include/logsys/control.h:40*, Implementáció: *src/shared/control.c:62*)

3.1.3. Aktív átviteli mód lekérdezése

Új adatátviteli mód megnyitása előtt célszerű megnézni, hogy van-e ütközés a korábban megnyitottakkal (bár ilyen esetben az LDC jelzi, hogy nem nyitható meg az átvitel). Az éppen aktív átviteli módot a következőképp kérdezhetjük le:

URB Control blokk:

Művelet	*Lekérdezés*
irány	IN
bRequest	3
wValue	0
wIndex	0
hossz	1

A visszaadott bájt jelzi az aktív átviteli módot (és 0 ha nincs megnyitva egyetlen átvitel sem). A lehetséges értékeket kigyűjtöttem egy enumerációba.

Implementáció

- Lekérdezés: `logsys_tx_get_active_func` (*include/logsys/control.h:43*, implementáció: *src/shared/control.c:67*)
- Enumeráció: `enum LogsysFunction` (*include/logsys/common.h:37*)

A kétféle JTAG módról a JTAG-ot tárgyaló fejezetben írok. Az egyes átviteli módok megnyitása/lezárása a róluk szóló fejezetben leírt kérésekkel történik.

3.1.4. Órajel generálás

Ha a CLK lábat nem használja egyetlen megnyitott átvitel sem, az LDC beépített négyszögjel-generátorának köszönhetően ki tudunk adni rajta egy órajelet (0,2 Hz és 8MHz között).

URB Control blokk:

Művelet	*Leállítás*	*Lekérdezés*	*Beállítás*
irány	IN	IN	IN
bRequest	4	4	4
wValue	0	0	periódusregiszter (PR)
wIndex	0x01	0x02	előosztás (E), 0x00
hossz	1	4	1

Leállításkor és elindításkor a sikerességet jelző (S) logikai változót kapjuk vissza. Lekérdezéskor pedig a következőt:

ofszet	0	1	2	3
mező	S	PR		E

3.1.5. Aszinkron reset

Ha nincs a RST lábat használó átvitel megnyitva, lehetőségünk van aszinkron bitbang módban vezérelni.

URB Control blokk:

Művelet	*Alacsonyra állítás*	*Magasba állítás*	*Lekérdezés*
irány	IN	IN	IN
bRequest	5	5	5
wValue	0	0	0
wIndex	0x00	0x01	0x02
hossz	1	1	1

Beállításnál a visszatérési érték 0, ha a RST láb használatban van, és >0, ha sikerült a beállítás. Lekérdezésnél a RST logikai értékét kapjuk vissza.

3.2. Átviteli módok

3.2.1. JTAG

Az első és talán legfontosabb implementált átviteli mód a JTAG. Ez a 0. interfész 0x01/0x82 enpoint-párját használja, valamint az LDC TDI, TDO, TMS, TCK és JTREF lábait. Ezen az interfészen keresztül lehetséges az LDC-hez illesztett elektronika konfigurálása és hibakövetése.

A JTAG megnyitásához két Control blokkot kell kiadnunk:
URB Control blokk:

Művelet	*JTAG inicializálás*	*Adatátvitel megnyitása*
irány	IN	OUT
bRequest	16	18
wValue	JTAG mód	0
wIndex	0	0
hossz	1	0

Ha az első kérés logikai hamisat ad vissza, megszakítjuk a műveletet. A "JTAG mód" paraméter leírása lejjebb található.

Megnyitott JTAG átvitel közben fel tudjuk deríteni a JTAG láncot (*Boundary Scan*), ehhez egy tapasztalati úton talált "mágikus sorozatot" kell leadnunk a 0x01 Bulk végponton, erre a `logsys_jtag_scan` függvény használható.

Az átvitel bezárásához a következő kérést használjuk:
URB Control blokk:

Művelet	*JTAG bezárás*
irány	OUT
bRequest	17
wValue	0
wIndex	0
hossz	0

Implementáció

- Megnyitás: `logsys_jtag_begin` (*include/logsys/jconf.h:6*, implementáció: *src/shared/jctrl.c:6*)
- Lezárás: `logsys_jtag_end` (*include/logsys/jconf.h:8*, implementáció: *src/shared/jctrl.c:28*)
- Lánc felderítés: `logsys_jtag_scan` (*include/logsys/jconf.h:13*, implementáció: *src/shared/jctrl.c:32*)

JTAG módok

A LDC kétféle módban tudja a JTAG interfészt kezelni:

- Visszaolvasás mód: ebben a módban a kiadott TDI értékekre kapott TDO választ visszaküldi USB transzferen keresztül a számítógépnek
- Ellenőrzés mód: ekkor a számítógép a TDI-vel együtt a várt TDO értékeket is elküldi, majd a transzfer után egy Control blokkal lekérdezi, hogy történt-e hiba (várttal nem egyező TDO érték)

Visszaolvasás módban működik pl. a Boundary Scan, mert ott az eszköz ID code-ját vissza kell adnunk a számítógépnek. SVF fájlok letöltése (lásd lejjebb, az "FPGA konfigurációs fájlok" részben) során azonban tudjuk, hogy milyen TDO értéket várunk (ez benne van az SVF állományban). Így ahelyett, hogy visszaolvasnánk USB-n keresztül, és a számítógép hasonlítaná össze, használhatjuk az LDC ellenőrzéses módját, ezáltal USB sávszélességet takarítva meg.

Az ellenőrzést a következő kéréssel végezzük:
URB Control blokk:

Művelet	*JTAG bezárás*
irány	IN
bRequest	20
wValue	0
wIndex	0
hossz	1

A módot megnyitott átvitel esetén is átválthatjuk:
URB Control blokk:

Művelet	*Lekérdezés*	*Beállítás*
irány	IN	OUT
bRequest	19	19
wValue	0	JTAG mód
wIndex	0	0
hossz	1	0

Implementáció

- Ellenőrzés: `logsys_jtag_check_error` (*include/logsys/jconf.h:19*, implementáció: *src/shared/jctrl.c:73*)
- Enumeráció: `enum LogsysJtagMode` (*include/logsys/common.h:42*)
- Mód lekérdezése: `logsys_jtag_get_mode` (*include/logsys/jconf.h:15*, implementáció: *src/shared/jctrl.c:62*)
- Mód beállítása: `logsys_jtag_set_mode` (*include/logsys/jconf.h:17*, implementáció: *src/shared/jctrl.c:69*)

JTAG adatátvitel

A TDI/TDO/TMS értékek megadása a 0x01/0x82 endpoint-párra küldött Bulk transzferekkel történik, egy speciális csomagformátumban:

TDI írás visszaolvasás módban:

	7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
1. oktett	0	*TMS*	0	bitek száma				
2. oktett	kiírandó TDI bitek							

TDI írás ellenőrzés módban:

	7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
1. oktett	0	*TMS*	0	bitek száma				
2. oktett	kiírandó TDI bitek							
3. oktett	várt TDO bitek							
4. oktett	ellenőrzés maszkja							

Az írás végén a TMS vonal *TMS* értékét veszi fel.

TMS írás visszaolvasás módban:

	7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
1. oktett	1	0	0	bitek száma				
2. oktett	kiírandó TMS bitek							

TMS írás ellenőrzés módban:

	7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
1. oktett	1	0	0	bitek száma				
2. oktett	kiírandó TMS bitek							
3. oktett	várt TDO bitek							
4. oktett	ellenőrzés maszkja							

Ekkor a kiírandó bitek a TMS vonalra kerülnek, nem pedig a TDI-re. Ez a funkció a TAP kontroller állapotának beállításához kell.[3]

RUNTEST visszaolvasás módban:

	7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
1. oktett	0	*TMS*	1	pulzusok száma				
2. oktett	pulzusok száma							

RUNTEST ellenőrzés módban:

	7. bit	6. bit	5. bit	4. bit	3. bit	2. bit	1. bit	0. bit
1. oktett	0	*TMS*	1	0x00				
2. oktett	0x00							
3. oktett	pulzusok száma							
4. oktett	pulzusok száma							

Ebben a módban TCK vonalra a megadott számú órajelpulzust lehet kiküldeni és a TMS vonal értéke a *TMS* bit lesz.[3] Ez a RUNTEST SVF parancsoknál hasznos. Ezt a módot nem használom, mivel a libxsvf elfedi előlem a RUNTEST parancsot.

Ellenőrzés módban a számítógépnek ugyanannyi bájtot kell visszaolvasnia, mint amennyit elküldött. A visszaolvasott bájtok közül csak minden második a hasznos TDO adat, a többi bájt értéke 0.[3]

FPGA konfigurációs fájlok

A libxsvf-nek köszönhetően SVF és XSVF formátumú fájlokat képes a program natívan kezelni. Azonban a BIT (és JED) formátumok nem nyíltak, így azok kezeléséhez a Xilinx egyik segédprogramját használva (iMPACT) át kell konvertálnom SVF-re (ugyanezt teszi egyébként a Logsys GUI is).

A segédprogrammal való kommunikációhoz a libc `popen` metódusát használom, ami a Unix pipe-okat használja (a Logsys GUI pedig temporális fájlokat hoz létre ugyanerre a célra). Azonban az iMPACT-nak nem lehet megmondani, hogy ne hozzon létre logfájlt, így hogy ne szemetelje tele a munkakönyvtárat, a szubprocessz elindítása előtt átlépek a `/tmp` könyvtárba, ami Unix-szerű rendszereken általában bárki által írható és nem perzisztens tároló. Ennek következtében azonban elveszítjük a relatív útvonalak használatának lehetőségét (hiszen mappát váltunk, és az így a `/tmp`-hez képest értelmeződne).

Valamint mivel a terminálból olvasáskor a parancsok paramétereit szóközzel választom el, problémás a szóközt tartalmazó útvonalak használata. Speciális esetként beletettem, hogy az aposztrófok vagy idézőjelek közötti szóköz ne törje az argumentumlistát, de pl. nem támogatom a "\ " formátumú escape-lést. Valamint ha egy útvonal tartalmaz aposztrófot ÉS idézőjelet IS, akkor nem reprezentálható.

Implementáció

- SVF letöltése: `logsys_jtag_dl_svf` (*include/logsys/jconf.h:25*, implementáció: *src/shared/jconf.c:202*)
- XSVF letöltése: `logsys_jtag_dl_xsvf` (*include/logsys/jconf.h:20*, implementáció: *src/shared/jconf.c:206*)
- BIT (vagy JED) konvertálása: `logsys_conv_bit2svf` (*include/logsys/jconf.h:35*, implementáció: *src/shared/jconf.c:210*)

3.2.2. SPI

Az LDC képes SPI (Serial Peripheral Interface) master módban működni (a csatlakoztatott programozható eszköz lesz a slave), a MISO, MOSI, CLK, RST és IOREF lábakat használva. Így egyszerű (az USART-nál könnyebben implementálható) soros adatátviteli összeköttetést tudunk létesíteni a számítógép és a fejlesztői kártya közt. Valamint a Logsys FPGA kártyákon található flash csipek is SPI interfészen írhatók, illetve ezen keresztül tudják az FPGA-t is konfigurálni.

Az implementációt a flash chipek felprogramozását végző Windowsos segédprogram (Logsys FLASH[6]) visszafejtéséből nyertem. A RC3 verziónak nem része, a következő kiadásban fogom megjelentetni.

Az adatátvitel történhet 8, 4, 2 vagy 1 MHz-es órajel mellett, ezenfelül többféle módot támogat, amit nem sikerült visszafejtenem. Az adatok átviteléhez a 0x01/0x82-es Bulk endpoint-párt használja.

Az átvitel a következő két kéréssel nyitható meg:
URB Control blokk:

Művelet	*Inicializálás*	*Átvitel indítása*
irány	IN	OUT
bRequest	32	34
wValue	$freq$	0
wIndex	mód	0
hossz	1	0

A $freq$ mezőben jobbról 0, 1, 2 vagy 3 db 1-es bit lép be attól függően, hogy a CLK frekvenciát 8, 4, 2 vagy 1 MHz-en akarjuk használni. Az IN kérés a sikerességet adja vissza.

Ezután az adatátvitel egy 5 bájtos header kiküldésével kezdődik a 0x01-es USB végponton:

ofszet	0	1	2	3	4
mező	Adatblokk hossza				0x01

Majd pedig az adatblokk következik, ugyanezen a végponton:

ofszet	0	1 ...
mező	parancs	adatok

Majd a 0x82-es végpontról szintén Bulk átvitellel visszaolvassuk a státuszkódot és a vett adatokat:

ofszet	0	1 ...
mező	státusz	adatok

Ezt a műveletsort végzi el nekünk a `logsys_spi_cmd` függvény.

Az átvitel bezárásához a következő kérést használjuk:
URB Control blokk:

Művelet	*SPI bezárás*
irány	OUT
bRequest	33
wValue	0
wIndex	0
hossz	0

Implementáció
(nem része a RC3-as verziónak. A sorszámok a `644e06153c1351cc038acfa3bf2212f86500aa48`-as commitra értendőek)

- Megnyitás: `logsys_spi_begin` (*include/logsys/serio.h:124*, implementáció: *src/shared/serio.c:67*)
- Lezárás: `logsys_spi_end` (*include/logsys/serio.h:126*, implementáció: *src/shared/serio.c:73*)

- Parancs küldés: `logsys_spi_cmd` (*include/logsys/serio.h:129*, implementáció: *src/shared/serio.c:77*)
- SPI órajel: `enum LogsysSpiSpeed` (*include/logsys/serio.h:53*)
- SPI mód (tartalma ismeretlen): `enum LogsysSpiMode` (*include/logsys/serio.h:61*)
- SPI parancsok: `enum LogsysSpiCmd` (*include/logsys/serio.h:64*)

3.2.3. USART

Az USART (Universal Synchronous/Asynchronous Receive & Transmit) adatátvitellel általános célú full duplex soros kapcsolatot létesíthetünk a számítógép és a céleszköz között. Az LDC képes hardveresen kezelni az USART átviteleket, a MOSI (TX), MISO (RX), CLK (csak USRT mód esetén) és IOREF lábakon keresztül.

Az átvitel indítása a következő kérésekkel történik:
URB Control blokk:

Művelet	*Inicializálás*	*Órajel megadása*	*Vonali kódolás beállítása*
irány	IN	OUT	OUT
bRequest	49	53	54
wValue	USRT mód	periódusregiszter	kódolás
wIndex	0	0	paritásbitek
hossz	1	0	0

Az IN által visszaadott bájt a kérés sikerességét jelzi. A vonali kódolás paramétereinek beállítására enumerációkat hoztam létre.

A portra való írás egy egyszerű Bulk átvitellel történik a 0x05-ös végponton. Olvasás esetén azonban figyeljünk arra, hogy a 0x86-os endpointról való olvasás első bájtja egy státuszbájt! Ez a státusz egy bitmező, jelentése a `LogsysUsartStatus` enumerációban található.

Az átvitelt ezzel a kéréssel zárhatjuk be:
URB Control blokk:

Művelet	*Bezárás*
irány	OUT
bRequest	50
wValue	0
wIndex	0
hossz	0

Illetve lehetőségünk van a soros port képességeinek lekérésére (minimális és maximális baudráta, órajel stb.)
URB Control blokk:

Művelet	*Képességek lekérdezése*
irány	IN
bRequest	48
wValue	0
wIndex	0
hossz	24

Ez egy ilyen struktúrát ad vissza:

ofszet	0	1	2	3	4	5	6	7	8	...
mező	órajel				előosztás	min. baud/s				...

ofszet	9	10	11	12	13 ... 21	22	23
mező	max. baud/s					major verzió	minor verzió

Implementáció

- Megnyitás: `logsys_usart_begin` (*include/logsys/serio.h:103*, implementáció: *src/shared/serio.c:36*)
- Lezárás: `logsys_usart_end` (*include/logsys/serio.h:105*, implementáció: *src/shared/serio.c:49*)
- Beolvasás: `logsys_usart_getstr` (*include/logsys/serio.h:114*, implementáció: *src/shared/serio.c:53*)
- Kiírás: `logsys_usart_putstr` (*include/logsys/serio.h:121*, implementáció: *src/shared/serio.c:63*)
- Vonali kódolás: `enum LogsysUsartEncoding` (*include/logsys/serio.h:41*)
- Paritás: `enum LogsysUsartParity` (*include/logsys/serio.h:48*)
- USART képességek: `logsys_usart_get_caps` (*include/logsys/serio.h:100*, implementáció: *src/shared/serio.c:32*), `struct LogsysUsartCaps` (*include/logsys/serio.h:30*)
- Státuszkódok: `enum LogsysUsartStatus` (*include/logsys/serio.h:21*)

3.2.4. BitBang soros I/O

A LDC képes a MISO, MOSI, CLK, RST és IOREF lábait használva szinkron BitBang típusú soros adatátvitelre. Ezzel szinte bármilyen adatátviteli protokoll szimulálható, azonban legfeljebb 1 kHz-es órajel mellett (a nagy USB adatforgalom miatt).

Az LDC vezérli a MOSI, CLK és RST lábakat, és a CLK fel- illetve lefutó éleire mintavételezi ezeket és a MISO lábat. Az órajel elindításához és az adatátvitel megkezdéséhez a következő két kérést kell elküldenünk:

URB Control blokk:

Művelet	*Inicializálás*	*Átvitel indítása*
irány	IN	OUT
bRequest	96	98
wValue	periódusregiszter	0
wIndex	előosztás	0
hossz	1	0

Az IN által visszaadott bájt jelzi a kérés sikerességét, és 0, ha már egy másik átvitel használja valamelyik kivezetést.

Az órajel-frekvenciát adatátvitel közben is megváltoztathatjuk:

URB Control blokk:

Művelet	*Órajel változtatás*
irány	OUT
bRequest	99
wValue	periódusregiszter
wIndex	előosztás
hossz	0

Átvitelt kezdeményezni egy 0x03 endpointra küldött `LogsysSerialLines` struktúrával lehet, majd a 0x84 végpontból olvasható ugyanilyen formátumban a mintavételezett érték. A szükséges bitmaszkokat az `enum LogsysSerialPin` tartalmazza. A struktúra kiosztása:

ofszet	0	1
mező	(00CR OI00)	(00cr oi00)

Az első oktett a CLK felfutó, a második a lefutó élén vett értékek és C=CLK, R=RST, O=MOSI és I=MISO.

Az átvitelt lezárni pedig a következőképpen lehet:
URB Control blokk:

Művelet	*Átvitel bezárása*
irány	OUT
bRequest	97
wValue	0
wIndex	0
hossz	0

Implementáció

- Megnyitás: `logsys_serial_begin` (*include/logsys/serio.h:90*, implementáció: *src/shared/serio.c:7*)
- Órajel változtatás: `logsys_serial_change_clk` (*include/logsys/serio.h:92*, implementáció: *src/shared/serio.c:17*)
- Lezárás: `logsys_serial_end` (*include/logsys/serio.h:94*, implementáció: *src/shared/serio.c:22*)
- Átvitel: `logsys_serial_send` (*include/logsys/serio.h:97*, implementáció: *src/shared/serio.c:26*)
- Vonalak állapota: `enum LogsysSerialLines` (*include/logsys/serio.h:10*)
- Bitmaszkok: `enum LogsysSerialPin` (*include/logsys/serio.h:18*)

3.2.5. Egyéb átviteli módok

Az LDC képes I^2C master és PIC ISP átvitelekre is, azonban ezekhez nem elérhető dokumentáció. Az I^2C átvitelhez létezett egy terminál app (az Interneten találni képet róla), azonban a futtatható állományt nem sikerült fellelnem, így ezeket nem tudtam beépíteni.

4. A driver felépítése

Maga a meghajtóprogram a `liblogsys-drv.so.0.0.1` osztott függvénykönyvtárban van. A hagyományos driverekkel ellentétben nem kernel objektum, hanem dinamikus library. Ezáltal a lehető legkisebb jogosultsági szinten, az őt éppen belinkelő alkalmazás szintjén fut, ami biztonsági szempontból előnyös. Továbbá nem kell minden kernelfrissítéskor újrafordítani, sőt, egyáltalán a Linux kernel megléte sem feltétel. Így bármilyen, LibUSB-t és libc-t tartalmazó platformon fordítható, például macOS-en.

Azonban ennek az alacsony jogosultságnak következtében nem tehetek meg néhány dolgot, pl. nem tudok device node-ot létrehozni a soros port számára, így azt is függvényhívásos interfészen keresztül kell használni.

4.1. Fájlstruktúra

A függvénykönyvtár a *src/shared* mappa fájljaiból fordul, valamint tartalmazza a libxsvf-et (hiszen az statikusan linkelt).

- *src/shared/control.c*: Ebben a fájlban találhatóak az *include/logsys/control.h*-ban deklarált függvények implementációi: a tápfeszültséggel, órajellel, aszinkron resettel és az LDC állapotleíróival kapcsolatos műveletek.
- *src/shared/jconf.c*: Itt az FPGA konfiguráció-letöltés implementációja található (erről bővebben lejjebb olvashatunk).
- *src/shared/jctrl.c*: Ebben a fájlban vannak az *include/logsys/jconf.h* egyéb függvényei (átvitel megnyitása/bezárása, módváltás stb.)
- *src/shared/serio.c*: Itt a soros adatátviteli módok (BitBang IO, USART, SPI stb.) megvalósítása, a *include/logsys/serio.h* által definiált módon.
- *src/shared/status.c*: A *include/logsys/status.h*-beli, státuszkódokat feldolgozó függvények, valamint itt található az órajel-generátor regisztereinek értékét előállító segédfüggvény (`logsys_create_clk_status`) is.
- *src/shared/usb.c*: Általános USB műveletek: LibUSB inicializálása/leállítása, eszköz megnyitása/elengedése, hotplug esemény elfogása stb. A függvények interfésze az *include/logsys/usb.h* fájlban van.
- *include/logsys/common.h*: A több helyen használt, nem kategorizálható enumerációk és struktúrák definíciói, illetve a Little Endian ⇔ Host Endian konverziót végző makrók találhatóak itt.
- *include/logsys/usb.private.h*: Végpont- és interfész-azonosítók gyűjteménye; csak belső használatú header, a drivert használó alkalmazások számára nem ajánlott a használata.

4.2. JTAG implementáció

Az SVF fájlok feldolgozását a libxsvf végzi. A feldolgozást a `libxsvf_play` függvény meghívásával indítjuk, aminek első paramétere egy `struct libxsvf_host` típusú változó. Ez a struktúra definiálja a libxsvf számára a JTAG vonalak beállításához hívandó függvény-pointereket. Ezeket a callbackeket tartalmazza a *src/shared/jconf.c* fájl. A függvény-implementációk neve szisztematikusan, a pointer nevéből egy `"lsvf_host_"` prefixszel képződik (azaz a `foo` metódust a `lsvf_host_foo` valósítaná meg). A továbbiakban ezekre a pointer nevével (a példámban a `foo`) hivatkozom. A struktúra tartalmaz még egy `struct udata_s`-re mutató pointert is, ahol tetszőleges felhasználói adatot tárolhatunk. Az én esetemben ebben található a LibUSB eszközre, illetve a nyitott SVF fájlra mutató pointer, a kiírandó, de még ki nem írt (LDC-nek el nem küldött) JTAG adatokat tároló puffer, valamint egy flag, hogy a legutolsó tranzakció (LDC-vel való kommunikáció) során lépett-e fel TDO összehasonlítási hiba.

4.2.1. `setup` és `shutdown`

A konfiguráció megkezdése előtt meg kell nyitni a JTAG átviteli módot a driver megfelelő függvényével, majd a letöltés befejeztével be kell zárni azt. Mivel egy SVF fájl feldolgozása a feladatunk, egyedül az ellenőrzéses módra lesz szükségünk, ez nagyban megkönnyíti az implementációt.

4.2.2. `getbyte`

A libxsvf ezt a függvényt hívja az SVF olvasásához. Nekünk csupán egy karaktert kell kiolvasnunk a `struct udata_s`-ben található fájlból, és visszaadni a libxsvf-nek.

4.2.3. `pulse_sck`, `set_trst` és `set_frequency`

Mivel az LDC nem rendelkezik TRST kivezetéssel, nem használja JTAG átvitel közben a soros kommunikáció CLK vonalát és nem támogatja a TCK frekvenciájának állítását, így ezeket a függvényeket nem kell implementálnom.

4.2.4. `report_*`

Ezek a függvények a felhasználó tájékoztatására szolgálnak. Főleg debug információkat írnak ki, ezért nem implementáltam (kivétel ez alól a `report_error`, amit hibaállapotban hív meg a libxsvf, és ennek hatására STDERR-re kiírom a hibaüzenetet).

4.2.5. `udelay` és `realloc`

A `udelay` hívás arra szolgál, hogy adott mikroszekundum időtartamig, a TMS-t megadott értékre állítva kiadjak néhány TCK pulzust. Erre szolgálna a RUNTEST mód, amit azonban relatív ritkasága és körülményes implementációja miatt nem használok, ehelyett ezt a `pulse_tck` függvénnyel (lásd lejjebb) oldom meg.

A `realloc` hívást változatlanul továbbítom a libc felé.

4.2.6. `pulse_tck`

Ez a JTAG konfiguráció "lelke". Nevével ellentétben nem csupán egy TCK pulzust ad ki, hanem rögtön be is állítja a kívánt TDI/TMS értékeket, sőt, a várt TDO-t is ekkor kapjuk meg (ha van).

Ezeket az adatokat eltároljuk a `struct udata_s`-ban található pufferbe, és ha szükséges, meghívjuk a `lsvf_flush_iobuf` függvényt, ami majd kiküldi az LDC-nek.

Visszatérési értékként a `struct udata_s`-ban található hibaflaget adjuk. Ezáltal ugyan a libxsvf késleltetve értesül a TDO ellenőrzési hibáról, de a tapasztalat azt mutatja, hogy ez nem okoz problémát.

4.2.7. `lsvf_flush_iobuf` és `lsvf_write_pack`

A `lsvf_flush_iobuf` függvény végzi a munka oroszlánrészét: az ő dolga a pufferelt TMS, TDI, TDO és TDO maszk adatokból az LDC csomagformátumának megfelelő kérést előállítani.

A függvény feladata "összevárni" a homogén adatokat (legfeljebb 8-at), majd őket egy keretbe tenni. Ezért először eldönti, hogy TMS vagy TDI írással valósítsa meg.

TMS írás

A TMS írás feltétele, hogy a TDI "Don't Care" legyen (a kódban -1), a TMS pedig 0 vagy 1. A `dbit` egy ring countert valósít meg, ami az éppen írandó bitet jelöli ki. A `data` tárolja a kiírandó adatot, a `chk` a várt TDO értékeket, a `cmask` pedig a TDO maszkot.

Amíg a TMS írás feltétele fennáll, `data`-ban eltároljuk a kiírandó TMS értéket (a `dbit` által kijelölt biten), valamint ha TDO nem "Don't Care", akkor őt eltároljuk `chk`-ben és bebillentjük `cmask` megfelelő bitjét.

Ha már összegyűjtöttünk 8 TCK pulzust, vagy már nem TMS írás történik, akkor `bulk_out`-ban létrehozunk egy keretet a kiszámított értékek számára.

TDI írás

A TDI írás feltétele, hogy a TDI 0 vagy 1 legyen, a TMS pedig az utolsó bit kivételével 0 (az utolsó bitnél lehet más is, az lesz az 1. bájt 6. bitje, lást 3.2.1/JTAG adatátvitel). A `dbit` itt is

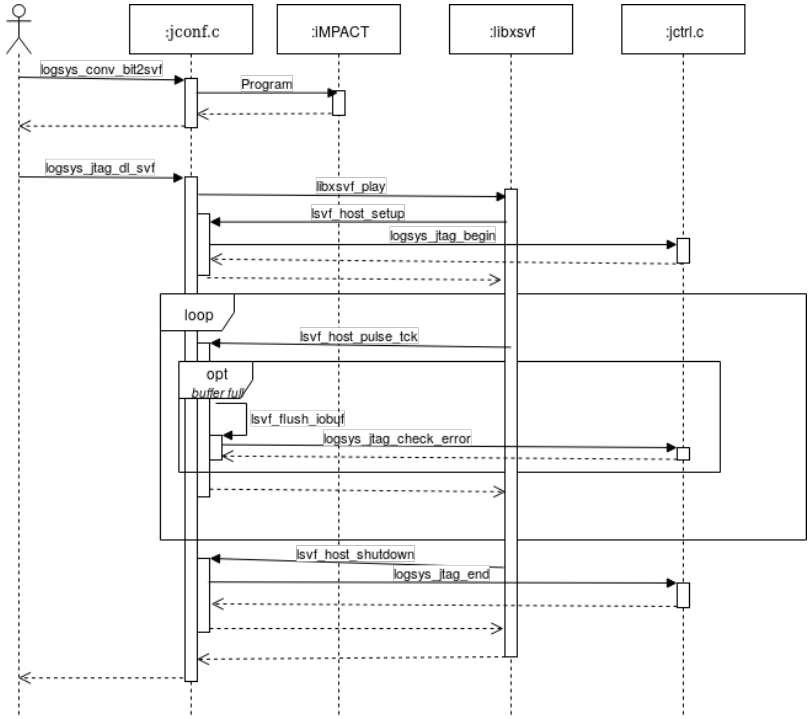
az éppen írandó bitet jelöli ki. A `data` tárolja a kiírandó adatot, a `chk` a várt TDO értékeket, a `cmask` pedig a TDO maszkot. Különbség csupán a `tmsAfter` változó jelenléte.

Amíg a TDI írás feltétele fennáll, `data`-ban eltároljuk a kiírandó TDI értéket (a `dbit` által kijelölt biten), valamint ha TDO nem "Don't Care", akkor őt eltároljuk `chk`-ben és bebillentjük `cmask` megfelelő bitjét.

Ha már összegyűjtöttünk 8 TCK pulzust, vagy már nem TDI írás történik, esetleg elértünk egy TDI+TMS íráshoz (ami csak a TDI írás utolsó bitjeként szerepelhet), akkor `bulk_out`-ban létrehozunk egy keretet a kiszámított értékek számára.

Tranzakció

Ezek után `lsvf_write_pack` feladata a keretek tömbjéből összeállítani egy Bulk átvitelt. Ehhez csupán sorosítania kell a tömböt, kiküldeni a 0x01-es végpontra, majd megnézni, történt-e hiba az összehasonlítás során.



7. ábra. UML szekvenciadiagram a "conf bit" parancs végrhajtásához hívott függvényekről

5. A parancssori alkalmazás felépítése

A konzolos teszt alkalmazás a `logsys-test` fájlnevet viseli. A programon belül különféle parancsokat adhatunk ki, amiket az LDC végre fog hajtani.

A program egyszerre pontosan egy csatlakoztatott LDC-t képes használni, és nem kezeli az LDC futás közbeni eltávolítását. Nem támogatja a terminál vezérlési szekvenciákat és a TAB-completiont sem. Valamint a driver által nyújtott lehetőségek csupán egy részhalmazát utilizálja.

A program forráskódja a *src/test/usbtest.c*.

5.1. Parancsok

A parancsok felépítése a következő: `kategória operáció [argumentumok]`. Az egyes kategóriák leírása következik lejjebb.

5.1.1. `status`

Ennek a kategóriának nincsenek operációi. Kiadáskor kiírja az LDC állapotát táblázatba formázva.

5.1.2. `clk`

Operációk:

- `status`: Órajel kimenet állapota
- `start <freq>`: Órajel elindítása `<freq>` Hz-en
- `stop`: Órajel kikapcsolása

5.1.3. `pwrlim`

Operációk:

- `get`: Túláram védelem küszöbértékének és a mérési korrekciós tényezőknek a lekérdezése.

5.1.4. `vcc`

Operációk:

- `get`: 5V kimenet állapota
- `on`: 5V bekapcsolása
- `off`: 5V kikapcsolása

5.1.5. `rst`

Operációk:

- `get`: RST kimenet állapota
- `on`: RST bekapcsolása
- `off`: RST kikapcsolása

5.1.6. `jtag`

Operációk:

- `scan`: JTAG Boundary Scan

5.1.7. `conf`

Operációk:

- `<formátum> <fájl>`: Konfiguráció letöltése. A `<formátum>` lehet `svf`, `xsvf`, `bit` vagy `jed`. A `<fájl>` egy **abszolút** elérési útvonal (lásd 3.2.1/FPGA konfigurációs fájlok)

5.1.8. `quit`

Hatására a program kilép.

5.2. További tesztprogramok

A parancssori példaprogram nem fedi le a driver teljes függvénykészletét, így a kimaradó műveletek tesztelésére további tesztprogramok érhetőek el a *src/test* mappában.

5.2.1. hotplug-test

Ez a program az USB hotplug eseménykezelést teszteli. LDC bedugásakor "`Logsys connected`", kihúzásakor "`Logsys disconnected`" üzenetet ír a képernyőre.

A program forráskódja a *src/test/hotplug.c*.

5.2.2. serio-test

Ez a program a BitBang soros I/O funkciót teszteli. Minden 3. órajelütemben felhúzza a MOSI, minden 5.-ben a RESET jelet, majd visszaolvassa a vonalak állapotát és formázva kiírja a terminálba.

A program forráskódja a *src/test/sio_fb.c*.

5.2.3. usart-test

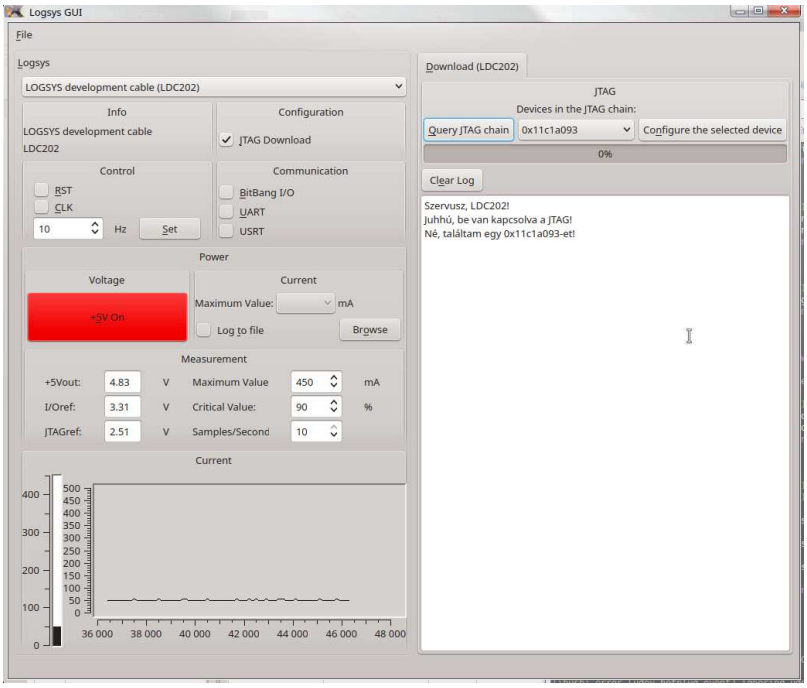
Ez a program az LDC USART átvitelét teszteli. A soros portra kiküld egy tesztüzenetet, majd beolvassa a rá kapott választ.

A program forráskódja a *src/test/uarttest.c*.

6. Lehetőségek a továbbfejlesztésre

Kezdő Linuxosoknak ijesztő lehet a parancssoros interfész, sokkal barátságosabb lenne egy Logsys GUI-ra hasonlító felület. Erre jelenleg két megközelítés jöhet szóba:

- Wacha Gábor Qt-os grafikus felületének integrálása. Itt szintén LibUSB-t és libxsvf-et használ, mint az én programom. Ehhez hozzá is kezdtem: az eredeti, hatékonytalan JTAG implementációt lecseréltem a saját, gyorsabb függvényemre. Azonban a teljes integráció előtt egy rendes C++ osztálystruktúrába kéne csomagolnom a driverem C-s függvényeit.
- Kiss Benedek Linux-kompatibilis .NET-es (Mono) megoldása. Itt is szükség lenne egy C ⇒ C# rétegre (ez azonban bonyolultabb lenne egy fokkal, mert P/Invoke-ot és managed/unmanaged Marshal-t kell hozzá használnom). Valamint a teljes támogatottság érdekében az egész Logsys GUI-t portolnom kéne .NET Framework-ről .NET Core-ra (vagy legalább .NET Standard-ra).



8. ábra. Wacha Gábor Qt alapú Logsys GUI reimplementációja, a saját módosításaimmal

Valamint fenntarthatósági szempontból sokkal szerencsésebb lenne, ha ugyanaz a kódbázis lenne Windowson, mint az összes többi platformon. Az általam létrehozott drivert kis ráfordítással (jelenleg egyedül a `usleep` függvény hiányzik) lehetne fordítani Windowson (a MinGW-t felhasználva), és a LibUSB is elérhető a rendszerre. Erre is megoldást nyújthatna a fentebb említett két projekt, hiszen mint a Qt, mint a .NET Core cross-platform támogatottságú.

További hiányosság a Windowsos rendszerhez képest, hogy nem hoz létre az LDC-hez COM portot (Unix rendszerben 'tty'), így más terminál-programokkal nem interkompatibilis. Erre Wacha Gábornak volt megoldása: egy Linuxos kernel driver. Ez azonban más Unixokon nem használható, és saját elmondása alapján nem is túl stabil. Illetve mivel ez egy tanulóplatform, nem feltétlen van szükség más terminálemulátorok használatára, ha a beépített jól működik. A kernel UART driver megmaradhat opcionális modulként.

Szeretném továbbá a nem dokumentált átviteleket implementálni; ehhez már el is kértem Raikovich Tamástól az LDC firmware-jét, illetve Fehér Bélától az I^2C terminálprogramot.

Ezen felül szeretném támogatni a MiniRISC rendszert. A MiniRISC egy tanszéki fejlesztésű, oktatási célú egyszerű softcore CPU, kvázi a Logsys ökoszisztéma része, amit szintén alkalmaznak a kari oktatásban. Azonban a MiniRISC IDE nevű fejlesztőeszköz sem működik Linux alatt, de a parancssoros assembler igen. Így a driveremmel le tudom tölteni a MiniRISC CPU-t az FPGA-ra,

le tudom fordítani az assembly programomat, és fel tudom vele konfigurálni a softcore processzort. Azonban ezzel csupán "Fire and Forget" módban tudok kódot futtatni rajta: feltöltöm a kódot, elkezd futni és onnantól nem látok bele a futásba. Ezzel szemben az IDE képes a futás felfüggesztésére, breakpointok kezelésére, single-step debuggolásra, a memória, a regiszterek és a perifériák olvasására és írására. Ezügyben felvettem a kapcsolatot a MiniRISC IDE eredeti fejlesztőjével, Fejér Attilával, aki rendelkezésemre bocsájtotta a kódot. A továbbiakban ezt kell modernizálnom és platformfüggetlenítenem.

Hivatkozások

[1] LOGSYS – Beágyazott rendszerek fejlesztői környezete
Fehér Béla, Raikovich Tamás, Laczkó Péter
http://logsys.mit.bme.hu/sites/default/files/page/2009/09/logsys_beagyazott_rendszerek_fejlesztoi_kornyezete.pdf

[2] LOGSYS SPARTAN-3E FPGA KÁRTYA FELHASZNÁLÓI ÚTMUTATÓ
http://logsys.mit.bme.hu/sites/default/files/page/2009/09/LOGSYS_SP3E_FPGA_Board.pdf

[3] Hatékony alkalmazásfejlesztés FPGA környezetben
Raikovich Tamás

[4] USB Capture Setup
WireShark Wiki
https://wiki.wireshark.org/CaptureSetup/USB

[5] XILINX ISE telepítése és használata Linux (64 bit) alatt
Csókás Bence Viktor
http://users.hszk.bme.hu/~cb1719/dlc/Xilinx_Linux.pdf

[6] Logsys FLASH
http://logsys.mit.bme.hu/sites/default/files/page/2009/09/LogsysFLASH_20121015.zip