



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs Rendszerek Tanszék

DSPController

Teljes dokumentáció

ÁLTALÁNOS INTERFÉSZ ILLESZTÉSE
JELFELDOLGOZÓ PROCESSZORHOZ

ÖNÁLLÓ LABORATÓRIUM 2.
ZÁRÓJEGYZŐKÖNYV

v1.1

Készítette
Simon Tibor

Konzulens
dr. Orosz György

2014. január 16.

Tartalomjegyzék

1. Bevezetés	4
1.1. DSPController	4
1.2. A dokumentum használatáról	5
1.3. Online források	5
2. Hoszt egység	6
2.1. ADSP-21364	6
2.1.1. Kártya felépítése	6
2.1.2. Csatlakozási felület	7
2.2. Szoftver	8
2.2.1. Hoszt szoftver feladatai	8
2.2.2. Működési blokkok	9
2.2.3. Beállítási lehetőségek	9
3. DSPController interfész	11
3.1. Hardver	11
3.1.1. Tervezés	11
3.1.2. Eszköz programozása	12
3.1.3. Alkatrészek listája	13
3.2. Firmware	14
3.2.1. Firmware feladatai	14
3.2.1.0.1. Beviteli eszközök - Input	14
3.2.1.0.2. Kommunikáció	16
3.2.1.0.3. Kimeneti eszközök - Output	17
3.2.2. Firmware blokkvázlata	18
4. SPI protokoll	20
4.1. Tervezési szempontok	20
4.2. AVR és SHARC - közös nevező	20
4.3. Protokoll felépítése	21
4.3.1. Típusok	22
4.3.2. Csomagolás	24
5. DSPController API v1.0	25
5.1. Típusdefiníciók	25

5.2. Események	25
5.3. Core API függvények	27
5.3.1. void DSPController_init_default(void)	27
5.3.2. void DSPController_init(int code)	28
5.3.3. void DSPController_tick(void)	29
5.3.4. void DSPController_flush(void)	29
5.3.5. void DSPController_led(Led leftLed, Led rightLed)	30
5.3.6. void DSPController_lcd_top(const char* format, ...)	30
5.3.7. void DSPController_lcd_bottom(const char* format, ...)	31
5.3.8. void DSPController_lcd(char line, const char* format, ...)	31
5.3.9. DIP DSPController_get_dip(void)	32
5.3.10. Event DSPController_get_event(void)	32
5.3.11. Encoder DSPController_get_encoder(char encoder)	33
5.4. Kiegészítő függvénycsomagok	34
5.4.1. Assembler	34
5.4.1.1. Assembler használata	34
5.4.1.2. Assembler API	36
5.4.1.2.1. DSPC_CONV_TYPE szimbólum	37
5.4.1.2.2. void DSPController_assembler_engage(void)	37
5.4.1.2.3. DSPC_CONV_TYPE DSPController_assembler_get_last(void)	37
5.4.1.2.4. DSPC_CONV_TYPE DSPController_assembler_get_current(void)	38
5.4.1.2.5. DSPC_CONV_TYPE DSPController_assembler_finish_number(void)	38
5.4.1.2.6. unsigned char DSPController_assembler_is_new_number(void)	38
5.4.2. Float to String Converter	38
5.4.2.1. Konvertáló függvény	39
5.4.2.1.1. char* DSPC_FTS(DSPC_CONV_TYPE f, unsigned char precision)	39
6. Példa projektek	40
6.1. Projektek felépítése	40
6.1.1. Project fájlrendszer	41
6.1.1.1. ad1835.h	41
6.1.1.2. tt.h	41
6.1.1.3. DSPController.h	41
6.1.1.4. GLUE.h	41
6.1.1.5. IRQprocess.c	41
6.1.1.6. SPORTisr.c	41
6.1.1.7. init1835viaSPI.c	42
6.1.1.8. initDAI.c	42
6.1.1.9. initPLL.c	42
6.1.1.10. initSPORT.c	42
6.1.1.11. main.c	42
6.1.1.12. DSPController.c	43

6.1.1.13. DSP.c	43
6.1.1.14. INTERFACE.c	43
6.2. Módosítandó és nem módosítandó fájlok	43
6.3. Tipikus programozás menete	43
6.4. Példakódok működése	44
6.4.1. Bare Minimum App	44
6.4.1.1. GLUE.h	45
6.4.1.2. DSP.c	45
6.4.1.3. INTERFACE.c	46
6.4.2. Demo App	47
6.4.2.1. Működés	47
6.4.2.2. Szoftveres fogások	48
6.4.2.2.1. GLUE változók használata	48
6.4.2.2.2. Menu pointer	49
6.4.2.2.3. Állapotváltások közötti eseménytörlés	49
6.4.2.2.4. Állapotok elnevezése	50
6.4.2.2.5. GLUE vagy nem GLUE változó	50
7. További fejlesztési lehetőségek	51
7.1. Ismert hibák	51
7.2. Fejlesztési lehetőségek	51
Irodalomjegyzék	52

1. fejezet

Bevezetés

1.1. DSPController

A BME VIK MIT tanszékén a DSP laborban két jelentős, elsősorban hangfeldolgozásra használható jelfeldolgozó rendszer található. Az egyik az Analog Devices által gyártott DSP kártyák (Blackfin és SHARC jelprocesszorokkal), a másik a Soundart Chameleon-ja [6]. Mindkét eszközöknek megvannak a jó és rossz tulajdóságai.

Az Analog Devices kártyák könnyen programozhatóak, jól támogatott fejlesztőkörnyezetük van, akár fixpontos, akár lebegőpontos kártyákat használunk. Egyedüli hátrányuk a futás közbeni felhasználói beavatkozás korlátozott lehetősége. Mindkét típusú kártyán csupán néhány nyomógomb és led áll rendelkezésre. A lehetőségek erősen korlátozottak.

A Chameleon egy kész programozható zenei termék. Egyedi a piacra (ami nem véletlen, hiszen használata viszonylag bonyolult). Egy szabadon programozható, kétprocesszoros rendszerről van szó, amit egy remek felhasználói interfésszel láttak el. A két processzor közül az egyik a jelfeldolgozásért felel (Freescale 56303), a másik általános processzor minden más feladatért (DSP programozása, MIDI, felhasználói interfész kezelés, sorosport kezelés). A jól kezelhető interfészhez rossz programozási környezet és nyelv társul.

Az általános processzoron keresztül lehet minden egységet elérni a rendszerben. A processzoron egy RTOS szoftver fut. Ez az első nehézség, de rövid tanulás után megszokható a kezelés. A másik, égetőbb probléma, hogy a jelfeldolgozásért felelős jelprocesszort csak assembly nyelvel lehet programozni. Ehhez azonban ismerni kell a jelprocesszor teljes belső felépítését legalább alapszinten, hogy a legegyszerűbb feladatokat meg tudjuk valósítani. A rendszer további hátránya, hogy nem lehet hagyományosan debuggolni, azaz a programot futás közben megállítani.

Az előzőekből kiderült, hogy aki fejleszteni szeretne DSP eszközökön, annak kompromisszumot kell kötnie kényelmes programozás és kényelmes futás alatti beavatkozás között.

Innen jött a DSPController ötlete. Mi lenne, ha ötvözni lehetne az Analog Devices kár-

tyák programozási megoldásait a Chameleon felhasználói interfészével? Mivel a Chameleon rendszerébe nem lehet javítani a programozhatóságot, ezért készítünk az Analog Devices kártyákhoz egy kiegészítő hardvert, ami egy bővebb felhasználói interfészt ad a kártyához.

A következő fejezetekben megismerkedük a DSPController minden egyes részegységével és azok működésével.

1.2. A dokumentum használatáról

A zárójegyzőkönyvben a következő elnevezési konvenciók használatosak.

Hoszt	Analog Devices ADSP-21364 fejlesztői kártya és az azon futó program.
Eszköz	Maga a DSPController kártya.
Felhasználó	Az az ember, aki használja a hosztot és az eszközt.
Programozó	Az a felhasználó, aki a hoszton futó kódot fejleszti.

A nagy terjedelemre való tekintettel létezik egy gyors kezdési lépésekkel tartalmazó dokumentum a **DSPController - Gyors kezdési útmutató** [7], ami a kezdeti lépésekben segíti a programozót.

1.3. Online források

Az egyes fejezetekben között online elérhető forráskódok jegyzéke:

Firmware	3.2	https://github.com/tiborsimon/DSPController-Firmware
Hardware	3.1.1	https://github.com/tiborsimon/DSPController-Hardware
Bare Minimum App	6.4.1	https://github.com/tiborsimon/DSPController-BareMinimumApp
Demo App	6.4.2	https://github.com/tiborsimon/DSPController-DemoApp

Minden forráskód a GitHub rendszerén van tárolva, bárki által szabadon módosítható akár online, akár offline is.

2. fejezet

Hoszt egység

A rendszer kifejlesztésének az első lépése a hoszt rendszer kiválasztása volt. A választás hamar az Analog Devices SHARC ADSP-21364 kártyájára esett, mivel programozási szempontból egy lebegőpontos jelprocesszor egyszerűbben használható, mint a fixpontos társai.

2.1. ADSP-21364

Az ADSP-21364 fejlesztői kártya egy lebegőpontos jelfeldolgozó rendszer. Alapvetően zenei jelfeldolgozásra készült, de a változatos interfészeinek és a jelfeldolgozási kapacitásának köszönhetően alkalmas más jellegű feladatok ellátására is.

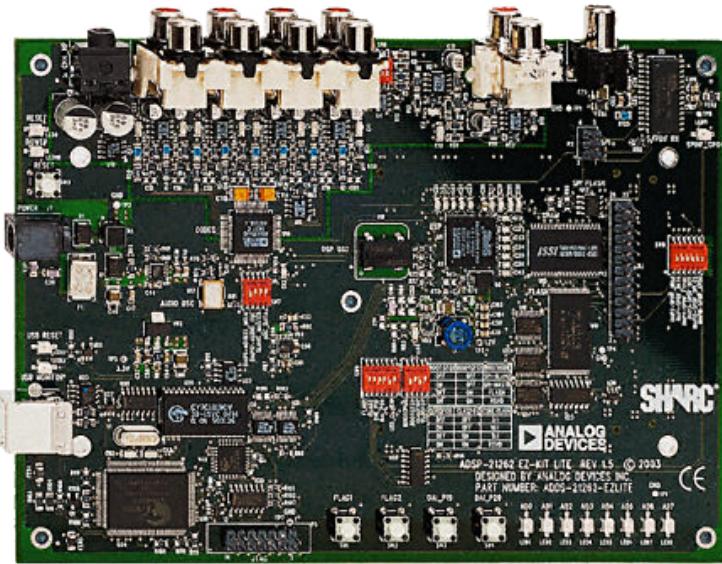
A kártyáról részletes információk az ADSP-21364 EZ-Kit Lite Manual-ban találhatók [4].

2.1.1. Kártya felépítése

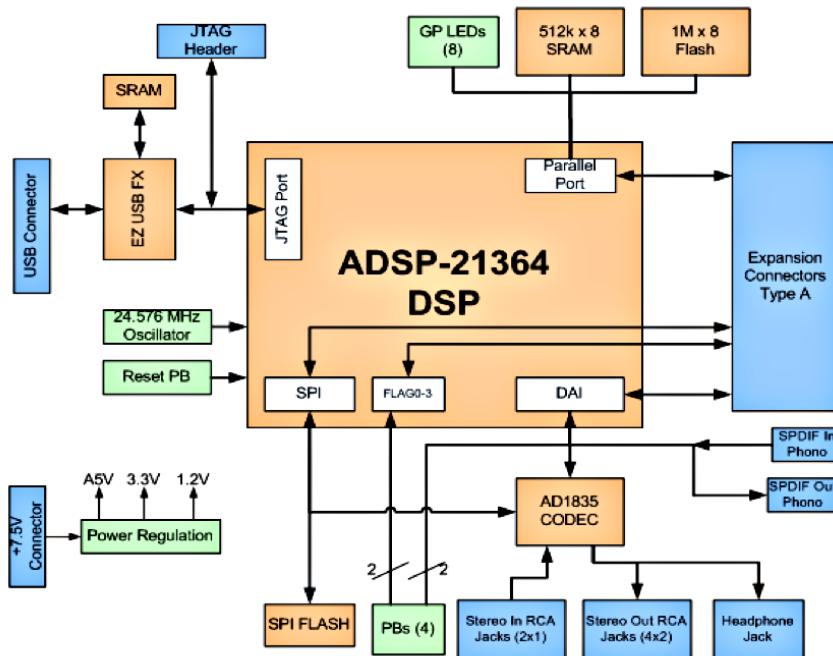
A kártya központi egysége az Analog Devices ADSP-21364 SHARC [5] 32 bites, lebegőpontos, 333 MHz frekvencián üzemelő jelfeldolgozó processzor. Memóriákat tekintve jól felszereltnek mondható: 3 Mbit RAM és 4 Mbit ROM áll rendelkezésre. A DSP egyedi architektúrájának köszönhetően egyszerűen illeszthetők hozzá külső egységek.

A kártyán a jelprocesszoron kívül megtalálhatóak a fejlesztést és a használatot segítő perifériák (2.2. ábra).

- Programozásért és debuggolásért felelős USB port, és az azt kiszolgáló eszközök.
- Külső memóriák (SPI FLASH, párhuzamos porton elérhető SRAM és FLASH).
- AD1835 audio codec. Egy sztereó bemenet és 4 sztereó kimenet.
- 4 darab nyomógomb és 8 darab LED.
- Jelprocesszor lábait kivezető csatlakozók.



2.1. ábra. ADSP-21364 fejlesztői kártya

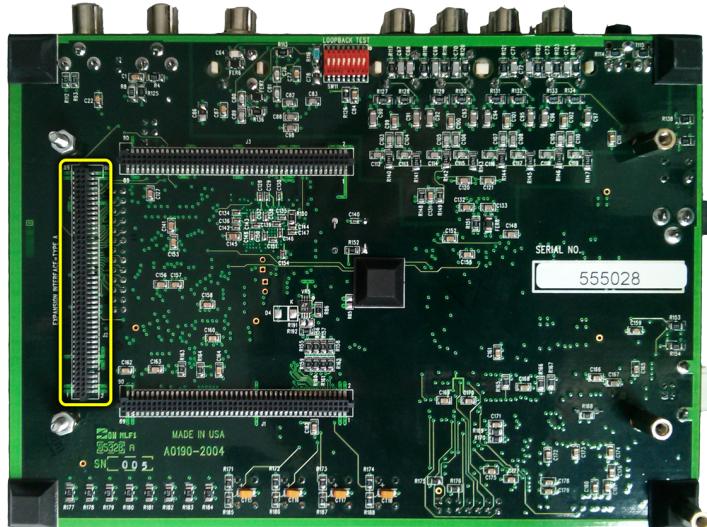


2.2. ábra. ADSP-21364 fejlesztői kártya blokkvázlata

2.1.2. Csatlakozási felület

A kártyán található csatlakozók lehetőséget biztosítanak külső hardverek csatlakoztatására. A 2.3. ábrán látható minden csatlakozó. A DSPControllerhez használt csatlakozó ki van emelve. Ez az egyetlen olyan csatlakozó, amire ki van vezetve az SPI busz és az általános GPIO-ként használható jelprocesszor lábak, ráadásul pont kényelmes oldalán van a kártyának.

A specifikáció kezdeti szakaszán tisztáztuk, hogy a hoszt és az interfész egység SPI



2.3. ábra. *ADSP-21364 fejlesztői kártya csatlakozói*

buszon fog kommunikálni. Ehhez a szükséges jelek elérhetőek a kiválasztott csatlakozón. Slave Select jelnek egy általános GPIO lábat (*DAIP15*) választottunk, hogy minimálisra csökkentsük az SPI ütközés kockázatát.

Az interfész egység a meglévő csatlakozóhoz tartozó furatokon keresztül rögzül a SHARC kártyához.

2.2. Szoftver

A hoszt egységen fut a DSPController programozó által felhasználható (és módosítható) kódja. Az interfész eléréséhez a DSPController API áll rendelkezésre, aminek a tárgyalására egy későbbi fejezetben kerül sor (5. fejezet).

Az API magában tartalmazza a DSPController driver programját. A driver feladata, hogy az alacsony szintű információcserét lebonyolítsa, a küldendő adatokat előkészítse az átvitelre, és a kapott adatokat könnyen elérhetővé tegye az API számára, ami egy magasszintű hozzáférést nyújt majd a programozónak.

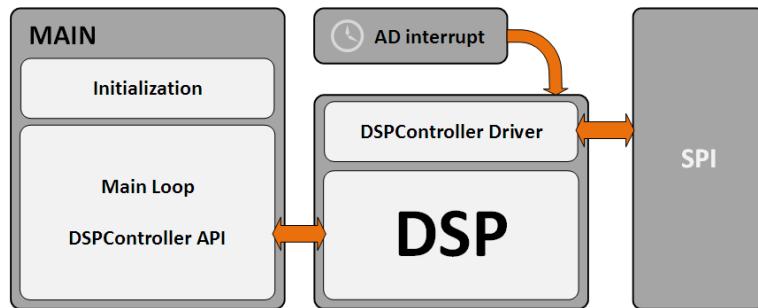
2.2.1. Hoszt szoftver feladatai

A DSPController hoszt egységen futó programjának a fő feladata, hogy kezelje a SHARC jelprocesszor SPI hardver modulját, és az azon keresztül történő adatforgalmat feldolgozza és a DSPController API-n keresztül elérhetővé tegye a programozó számára.

A hoszt egység C nyelven íródott. A kezdetektől fogva a lehető leghordozhatóbb forráskód készítése volt a szempont. Ez részben kényelmi szempontokból volt így. A szoftver fejlesztésének az első szakaszában a SHARC fejlesztői kártyát egy Arduino [1] emulálta. Az Arduino-n kifejlesztett első verzió portolva lett a SHARC kártyára, majd a megfelelő környezetfüggő kódrészletek átirása után a fejlesztés egy félkész szoftverrel folytatódott.

2.2.2. Működési blokkok

A 2.4. ábrán látható a hoszt egységen futó teljes szoftverrendszer blokkvázlata. Látható, hogy az alap jelfeldolgozó program két fő egységből áll: inicializálásból és az analóg-digitális konverter megszakításában futtatott jelfeldolgozó kódból. Ehhez a két egységhez épül be a DSPController használatához szükséges kód.



2.4. ábra. Hoszt szoftver felépítése

A DSPController hoszon futó kódja is két részből áll. Az egyik az alacsonyszintű, megszakítás által hívott, SPI hardvert kezelő DSPController Driver, a másik a főciklusban futtatott DSPController API-t használó kód.

A két részegység futási környezetéből következtetni lehet a végrehajtási idejükre.

A DSPController Driver egy magas prioritási szinten futó, gyors végrehajtási idejű kódrészlet, aminek az egyedüli feladata, hogy az SPI hardverre küldjön, vagy az SPI hardverről fogadjon, és további feldolgozásra előkészítsen adatokat. Ezt a lehető leggyorsabban kell tegye, hogy ne vegyünk el felesleges processzoridőt a lényegi jelfeldolgozási feladattól.

A főciklusban futó, DSPController API-t használó kód cikluson belüli futási ideje tetszőleges lehet. Az időkritikus programrészletek megszakítással jutnak érvényre. Tipikusan ez a kód felelős a DSPController interész működtetéséért, "*emberi sebességen*". A működtetési lehetőségeket a későbbi 6. fejezet tárgyalja részletesen.

2.2.3. Beállítási lehetőségek

Mivel a DSPController nem időkritikus működésű, ezért minden AD megszakításban felesleges ellenőrizni az SPI hardver állapotát (érkezett-e új adat vagy írható-e új adat rá), hiszen az esetek döntő többségében a válasz nemleges lenne. Ezért a DSPController Driver csak előre beállított gyakorisággal végzi el a feladatát. Ez a gyakoriság a jelfeldolgozó rendszer mintavételi frekvenciáról függ, és úgy lett beállítva, hogy megtalálja az egyensúlyt a leheteő legkevesebb overhead és az érezhetően késleltetésmentes reakcióidő között.

A következő kódrészlet tartalmazza a programozó által beállítható paramétereket és a jelenlegi verzióban található alapbeállításokat, amiket a DSPController Driver és a DSPController API használ. Néhány paraméter nem lehet világos elsőre, tisztázásuk a DSPController API fejezetében (5. fejezet) esedékes.

Az alapbeállítások módosítása csak abban az esetben javasolt, ha a programozó tisztában van a módosítások hatásával.

2.1. példakód. Felhasználó által beállítható paraméterek

```

1 //=====
2 // V A R I A B L E P A R A M E T E R S
3 //=====
4 #define DSPC_TICK_THRESHOLD_48 26
5 #define DSPC_TICK_THRESHOLD_96 52
6
7 #define DSPC_LED_COUNTER_MAX_48 4
8 #define DSPC_LCD_COUNTER_MAX_48 100
9 #define DSPC_DIP_COUNTER_MAX_48 20
10
11 #define DSPC_LED_COUNTER_MAX_96 10
12 #define DSPC_LCD_COUNTER_MAX_96 200
13 #define DSPC_DIP_COUNTER_MAX_96 50
14
15
16 #define DSPC_EVENT_BUFFER_SIZE 32
17
18 #define DSPC_CONV_TYPE float
19
20
21 #define DSPC_ENCODER_VELOCITY_THRESHOLD 300
22 #define DSPC_ENCODER_VELOCITY_MULTIPLIER 3

```

DSPC_TICK_THRESHOLD_48/96	AD megszakítások száma egy DSPController Driver aktivitás között 48/96 kHz-en.
DSPC_LED_COUNTER_MAX_48/96	LED oszlopok buszideje driver futási alkalmakban mérve 48/96 kHz-en.
DSPC_LCD_COUNTER_MAX_48/96	LCD kijelző buszideje driver futási alkalmakban mérve 48/96 kHz-en.
DSPC_DIP_COUNTER_MAX_48/96	DIP kapcsolók beolvasási gyakorisága driver futási alkalmakban mérve 48/96 kHz-en.
DSPC_EVENT_BUFFER_SIZE	DSPController API beérkező események tároló buffer mérete.
DSPC_CONV_TYPE	DSPController API-hoz tartozó kiegészítő <i>Assembler</i> (5.4.1) és <i>Float to String Converter</i> (5.4.2) visszatérési értékei. Alapértelmezetten szimpla pontosságú lebegőpontos ábrázolásban.
DSPC_ENCODER_VELOCITY_THRESHOLD	Időablak DSPController Driver futási alkalmakban mérve, amin belül az enkóderek tekerési eseményeihez többszörös értéket rendelünk.
DSPC_ENCODER_VELOCITY_MULTIPLIER	Enkóderek tekerésének többszörözési értéke.

3. fejezet

DSPController interfész

Miután a hoszt egység által nyújtott kapcsolódási lehetőségek tisztázásra kerültek, elkezdődhetett a DSPController egység hardverének a tervezése.

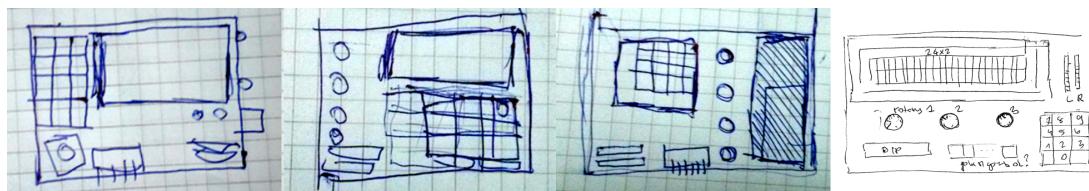
3.1. Hardver

3.1.1. Tervezés

A hardver tervezésének az első lépése a kívánt funkciók felmérése volt. Szerettük volna minél általánosan használhatóbbá tenni a kész hardvert. A rendszer az alábbi funkciókkal rendelkezik:

- 2x16 karakteres LCD kijelző
- 3 darab nyomógombos enkóder
- 4 darab funkciógomb
- 5 darab navigációs gomb
- numerikus billentyűzet
- 8 darab kapcsoló
- két oszlopos kivezérlésmérő

Az elrendezés megválasztása nem volt egyértelmű feladat, ebben Bank Balázs is segített nekünk (3.1. ábra).



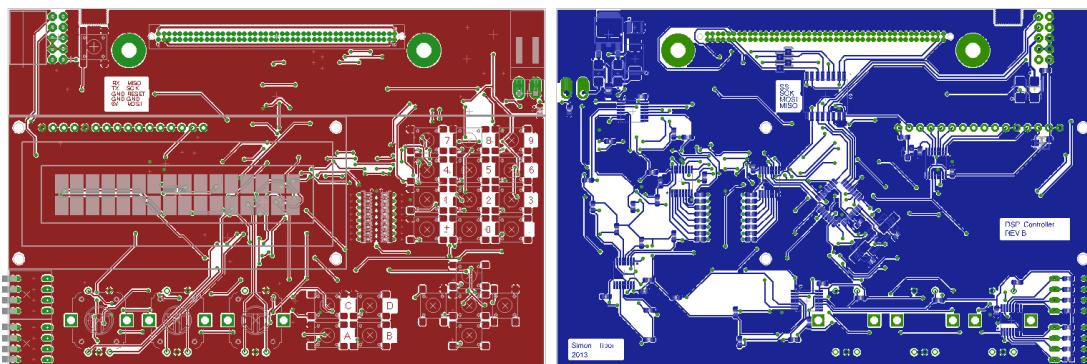
3.1. ábra. Vázlatok az elrendezéshez

Az elrendezés véglegesítése után a funkciókat kiszolgálni képes áramkör tervezése következett.

Központi vezérlőnek az Atmel ATmega328P [2] mikrokontrollert választottuk. Jól támogatott fejlesztőrendszer van, Arduino kompatibilis (később elvetettük az Arduino csomag használatát a optimalizálatlansága miatt), és könnyen beszerezhető.

Mivel a mikrovezérlőn közel sincsen annyi IO láb, mint ahány ki- és bemenetet kéne kezelní, ezért shift regiszteres bővítést alkalmaztunk. Soros-párhuzamos és párhuzamos-soros működésű shift regisztereket is használtunk a ki- és beolvasáshoz. Előnyük, hogy soros kommunikációval írhatóak/olvashatóak, ami kevés IO lábat emész fel, viszont előnyük a hátrányuk is egyben, ugyanis minden adatot léptetni kell, ami időbe kerül, és valamivel összetettebb programmal jár.

Fontos szempont volt, hogy a DSPController egység meghibásodása esetén az Analog Devices kártya minimálisan vagy semmennyeire ne sérüljön, ezért teljes galvanikus leválasztást, és külön tápvonalat alkalmaztunk. Galvanikus leválasztáshoz az Analog Devices ADUM integrált áramkörét alkalmaztuk [3].



3.2. ábra. *DSPController NYÁK alul- és felülnézet*

A specifikációk alapján elkészült a kapcsolási rajz (3.8. ábra), a NYÁK terve (3.2. ábra), majd a kész NYÁK is (3.3. ábra). A siker nem volt azonnali, az első NYÁK gyártási hibás volt, és egy rövidzár volt a fő tápvonalon. A NYÁK újra lett tervezve nagyobb szigetelési távolságokkal (*REV B*). A második áramkör már hibátlanak bizonyult, csupán egy SPI vonal bekötést kellett utólag módostani a csatlakozó kivezetésénél. minden hardveres gyártási fájl megtalálható a GitHub-on: <https://github.com/tiborsimon/DSPController-Hardware>

3.1.2. Eszköz programozása

A firmware feltöltése a hagyományos AVR felületen keresztül történik, az eszköz bal oldalán található csatlakozón keresztül. A csatlakozóra a soros kommunikációs vonalak is ki vannak vezetve debuggolási célokra. Nem készült külön kábel a rendszer programozásához, ugyanis a firmware-t frissíteni a legritkább esetekben szükséges. Programozáshoz a külön kábelekkel való összekötés javasolt.

A programozó csatlakozó kiosztása:

MISO	SCK	RESET	GND	MOSI
RX	TX	GND	GND	5V

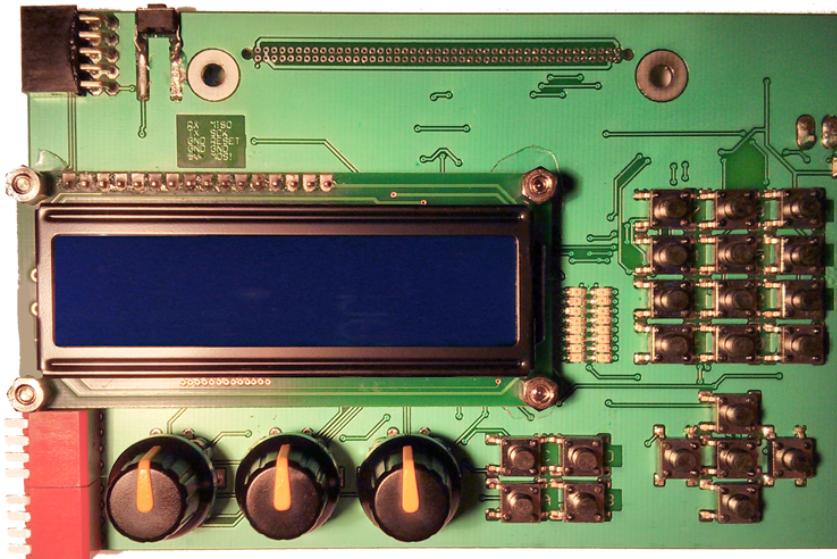
A jelenlegi konfiguráció felprogramozásához (v1.1) Atmel Studio 6.1 és AVR Dragon programozó lett használva.

3.1.3. Alkatrészek listája

Db	Érték	Eszköz	NYÁK Szimbólum
16	1k	RESISTOR0603-RES	R7..14, R17..24
8	1u	CAP0603-CAP	C5, C6, C8..13
34	10k	RESISTOR0603-RES	R1..6, R15, R16, R25..51
1	10k	TRIMPOTTC33X	TRIMPOT
1	16MHz	CRYSTALSMD	Y1
1	20	RESISTOR0603-RES	R50
2	22p	CAP0603-CAP	C2, C4
1	22u	CPOL-EUSMCB	C7
1	47u	CPOL-EUD/7343-31W	C1
4	74HC165	74HC165	IN_1, IN_2, IN_3, IN_4
3	74HC595	74HC595	SR_OUT1, SR_OUT2, SR_OUT3
1	100n	C-EUC0805K	C3
1	550	RESISTOR0603-RES	R52
1	7805DT	7805DT	VOLTAGE_REGULATOR
1	ADUM1401BRWZ	ADUM1401BRWZ	ADUM
1	ATMEGA168	ATMEGA168	AVR
1	BSS 138	MOSFET-NREFLOW	LCD_BACKLIGHT_FET
2	DIP04PIANO	DIP04PIANO	DIP1, DIP2
3	ENCODER_SW1-PEC12	ENCODER_SW1-PEC12	E1, E2, E3
1	M05X2RAF	M05X2RAF	PROGRAMMING_CONNECTOR
1	NSG 39-2	BUTTON_SMD	RESET
1	NSL39_2_CONNECTOR	NSL39_2_CONNECTOR	POWER_CONNECTOR
17	OF-SMD 2012 R/Y/G	LEDCHIPLED_0805	LED1..32
1	RC-1602-BB	TUXGR_16X2_R2	LCD
1	TFC-145-11-F-D-A	SAMTEC_TFC	CONNECTOR
4	SOLDERJUMPERNC2	SOLDERJUMPERNC2	SJ1, SJ2, SJ3, SJ4
21	TACTM-67N	BUTTON_SMD	A1, A2, A3, A4, A5, F1, F2, F3, F4, N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12
4	TACTM-67N	DIODESMA-ALT	D1, D2, D3, D4

Az egyetlen nehezen beszerezhető alkarész a SAMTEC csatlakozó volt. Ezt közvetlenül a SAMTEC-től rendeltük mintaként.

A gyártás és összeszerelés után a *B változat* beüzemelésre készen állt.



3.3. ábra. *DSPController kész áramkör csatlakozó nélkül*

3.2. Firmware

A firmware verziói

-
- **v1.0** Arduino alapok az egyszerű fejlesztés reményében. Problémák: az Arduino csomag nagyon optimalizálatlan, mind sebességbeli, mind kompatibilitásbeli problémák.
 - **v1.1** Arduino-ra épülő kód elvetése. Alapoktól újraírt, tiszta AVR-libc. Első teljes mértékben működőképes verzió.
-

A szoftverhez tartozó teljes forráskód és dokumentáció elérhető a GitHub hálózatán:
<https://github.com/tiborsimon/DSPController-Firmware>

3.2.1. Firmware feladatai

Az eszközön futó programnak három fő feladata van:

- Beviteli eszközök feldolgozása (nyomógombok, kapcsolók, enkóderek)
- Kiviteli eszközök meghajtása (LCD, kivezérlésjelző)
- Kommunikáció a hoszt egységgel

A feladatoknak megfelelő blokkvázlat a 3.6. ábrán látható.

3.2.1.0.1. Beviteli eszközök - Input

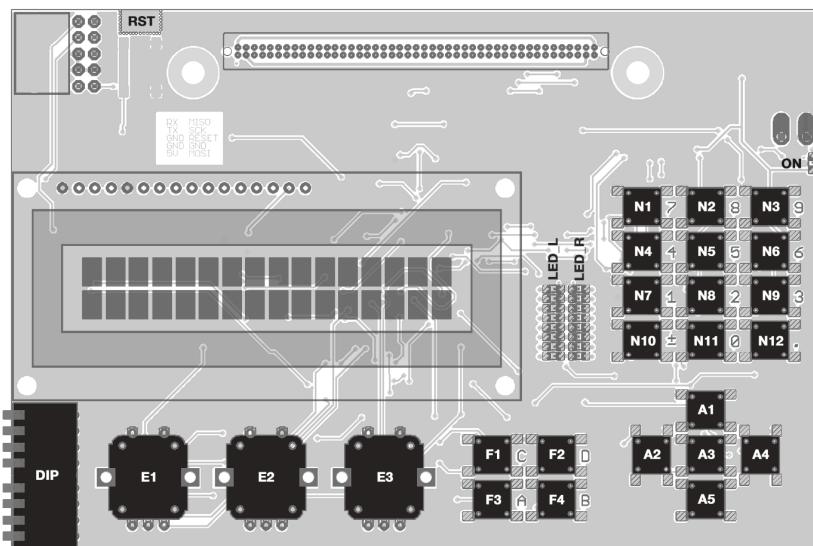
A DSPController eszközön 35 darab bevitelre alkalmas eszköz található.

- 24 darab nyomógomb, ebből három az enkódereken található
- 8 darab kapcsoló
- 3 darab, 24 lépéses inkrementális enkóder.

Mindegyik nyomógomb tetszőleges feladatra felhasználható a később tárgyalt DSPController API felhasználásával, viszont az áramkörön csoportokba vannak rendezve lehetséges funkciók szerint:

- numerikus billentyűzet
- navigációs gombok
- funkció gombok

Az egyes csoportokon belüli nyomógombok számozottak. A számozás a 3.4. ábrán láttható.



3.4. ábra. *DSPController eszköz felülnézet*

A később bemutatásra kerülő DSPController API-nál ezekkel az elnevezésekkel lehet majd hivatkozni az egyes nyomógombok eseményeire.

Az interfész egységen található beviteli eszközök mindegyike - az enkóderektől eltekintve - közvetve csatlakozik a mikrokontrollerhez shift regisztereken keresztül. A párhuzamos-soros átalakítók egymással párhuzamosan vannak a mikrovezérlőhöz kötve közös léptető jellel, hogy a kiolvasás sebességét minél kevésbé befolyásolja a shift regiszterek miatti soros kommunikáció.

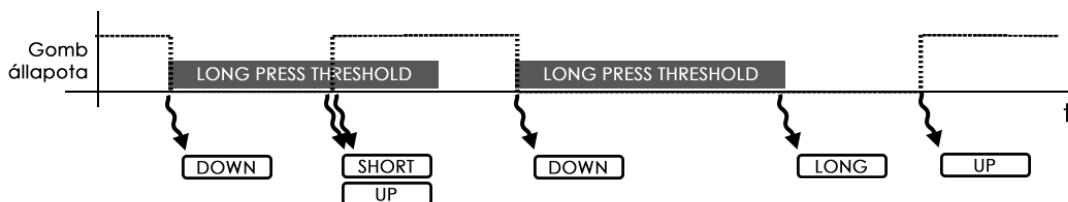
A bemeneti eszközök feldolgozását egy időzítő megszakításával ütemezzük. minden megszakításkor lefut egy algoritmus, ami a shift regisztereken keresztül lekérdezi a nyomógombok, kapcsolók, enkóder nyomógombok értékét, majd a megfelelő pergésmentesítés után

eseményeket generál az aktuális állapotoknak megfelelően.

Egy nyomógombhoz négy féle eseményt tud megkülönböztetni az algoritmus.

- **DOWN** - lenyomás
- **UP** - felengedés
- **LONG** - hosszú nyomás
- **SHORT** - rövid nyomás

A *lenyomás* és a *felengedés* egyértelmű, azonban a *rövid* és a *hosszú nyomás* magyarázatra szorul. minden nyomógombnál, minden lenyomás után elindul egy időzítő, ami méri, hogy mennyi ideig volt lenyomva az adott nyomógomb. Ha a felengedés pillanatában ez a számláló átlépett egy határt, akkor *hosszú nyomás* generálódik, ha nem, akkor *rövid nyomás*. Látható, hogy a két esemény kölcsönösen kizára egymást. Az is megfigyelhető, hogy a *rövid nyomás* mindig azonos időben generálódik a *felengedés* eseménnyel, míg a *hosszú nyomás* előbb keletkezik, mint a *felengedés*. A működés szemléltetése a 3.5. ábrán látható.



3.5. ábra. Gombok eseményei

Feldolgozási szempontból minden shift regiszteren beolvassott eszköz azonos módon van kezelve. Azonosan mindegyik eszközre minden esemény generálódik, de a nyomógombuktól eltérően a kapcsolókról csak a hosszú nyomás eseménye olvasható ki egy később tisztázott módon.

Az enkóderek közvetlenül csatlakoznak a mikrovezérlőhöz. Feldolgozásuk hasonló a nyomógombokéhoz, azzal a különbséggel, hogy itt nem eseményeket generál az algoritmus, hanem egy regiszternek növeli vagy csökkenti az értékét az enkóder tekerési irányától független. A regiszter a kiolvasáskor nullázódik.

3.2.1.0.2. Kommunikáció

A DSPController eszköz önmagától működésképtelen. Megjelenítendő adat nélkül a kimeneti eszközök nem üzemelnek, a bemeneti feldolgozási egység pedig periodikus kiolvasás nélkül egy idő után leáll a működéssel, mert megtelnek az eseményeket tároló bufferek.

A normális működéshez ezért szükség van a hoszt eszközre, ami adatokkal látja el, és periodikusan kiolvassa a generálódó eseményeket az interfész eszkövről. Az előző fejezetben tárgyaltuk, hogy a hoszt egység kialakítása miatt SPI buszon keresztüli kommunikáció

lehetséges a két eszköz között.

A fejezet elején láttuk, hogy az SPI jelek galvanikusan leválasztottan érkeznek a mikrovezérlőhöz. Magát a kommunikációt a hoszt egység vezérli, a DSPController slave-ként funkcionál. minden befejezett átviteli ciklus után az AVR SPI hardvere megszakítja a program futását egy *SPI átvitel kész* megszakítással. Ebben a megszakításban fut a kommunikációért felelős állapotgép, ami a következő fejezetben ismertetett SPI protokollt implementálja.

Az állapotgép egy kettős bufferból dolgozik, amit az esemény generáló algoritmus tölt fel eseményekkel. minden SPI átviteli ciklus elején (protokoll szerinti ciklusok, nem egy bájt átvitele az SPI buszon) a buffer olvasási és írási része helyet cserél, az olvasási részből dolgozik az SPI protokoll állapotgépe, az írási részbe teszi az új eseményeket az esemény generáló algoritmus.

Ha egy új érték kerül a buffer írási részébe, és az SPI buszon éppen nincs kommunikáció, akkor az SPI hardver kimenő regiszterébe bekerül az aktuális események száma, ami adatátvitel során rögtön a hoszt egységhez kerül, így az tudni fogja, hogy hány további átvitelt kell kezdeményezzen, hogy kiolvassa az eseményeket. (részletesen a következő fejezetben)

A kommunikációért felelős kód részlet a legmagasabb prioritású az egész rendszerben.

3.2.1.0.3. Kimeneti eszközök - Output

Az interfész egységen két kimeneti eszköz található.

- LCD kijelző
- LED-es kivezérlésjelző

Mindkét eszköz meghajtásához sok jelre van szükség, ezért a bemeneti eszközökhöz hasonlóan, itt is shift regisztereket alkalmazunk. Három 8 bites soros-párhuzamos átalakító pont képes ellátni a feladatot. A shift regiszterek egymás után vannak kötve, hogy a kimeneti eszközök minél kevesebb IO lábat foglaljanak le. Később kiderült, hogy ez a konfiguráció nem volt a legszerencsesebb, mivel bonyolultabb LCD-kezelő algoritmust igényel.

Az LCD kijelző minden hagyományos funkciójára képes, kivéve a saját szerkesztésű karakterek megjelenítése. Ez egy későbbi verzió bővítése lehet. A DSPController rendszerben egyszerre minden egy egész LCD sor frissíthető. Ez jelentősen egyszerűsíti a vezérlő algoritmus felépítését.

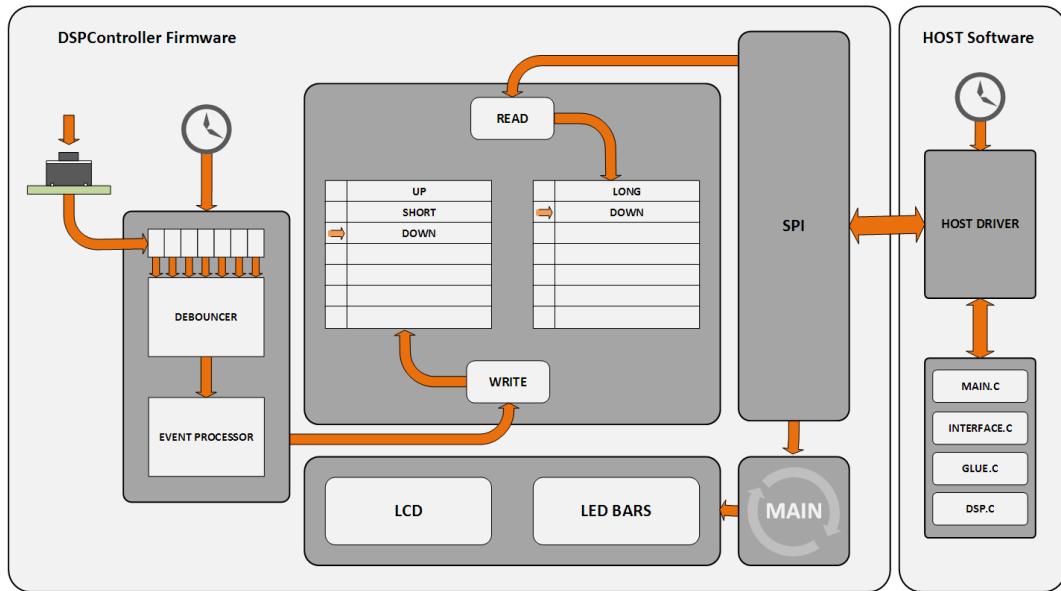
A LED-es kivezérlésjelzők működtetése egyértelmű. A shift regiszterekre kiküldött érték szerint gyulladnak ki az egyes LED-ek.

A kimenetek meghajtása alacsonyabb prioritású feladat, mint a bemenetek beolvasása és a kommunikáció kezelése, ezért a main() függvény végtelen ciklusában kapott helyet.

Ha a kommunikácos programrész egy kimenetre szánt adatot fogad, elteszi egy bufferbe, bebillent egy jelzőbitet, majd folytatja a feladatát. A főciklus, mikor épp nem megszakítás fut, a jelzőbők alapján kezeli a kimeneti eszközöket.

3.2.2. Firmware blokkvázlata

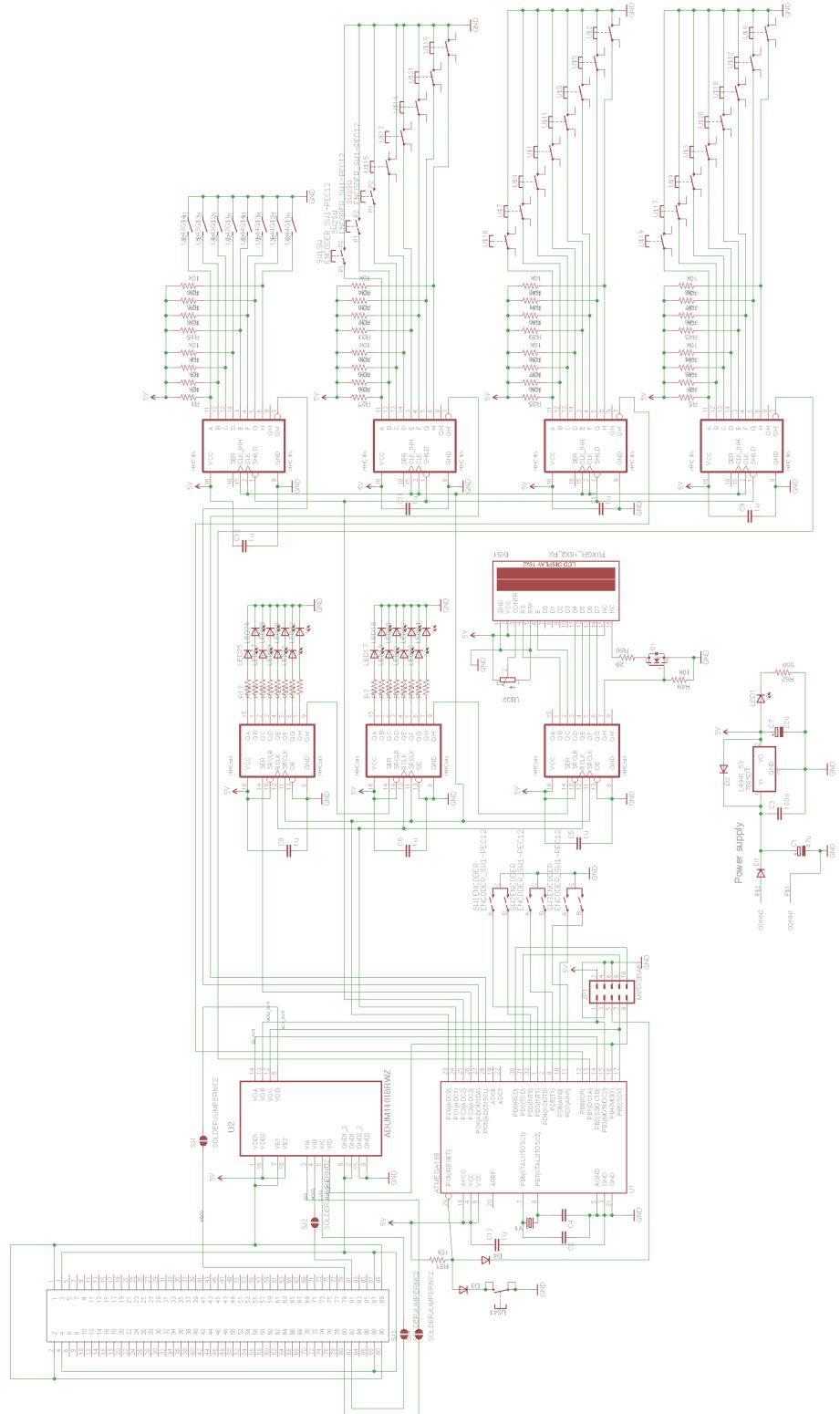
A 3.6. ábrán látható a DSPController szoftverének az blokkvázlata.



3.6. ábra. Firmware felépítése



3.7. ábra. DSPController működés közben



3.8. ábra. *DSPController kapcsolási rajz*

4. fejezet

SPI protokoll

A következő fejezet bemutatja az SPI protokoll tervezését és működését.

4.1. Tervezési szempontok

Az SPI átviteli protokolljának a tervezésénél az egyik legfontosabb szempont a gyorsaság volt. Az eszköznek nem kell túl szigorú határidőket betartania, hiszen egy "*emberi sebeségén*" működő interfész eszközről van szó. Azonban senki sem szeret egy gép reagálására várni.

A sebességet erősen korlátozza az eszközön található mikrokontroller SPI hardverének sebessége, ami a stabil működéshez maximum 4 MHz lehet. Ezen kívül a maximális egyszerre küldhető adathossz 8 bit.

Figyelembevéve a korlátozásokat a lehető legtömörebb kommunikációt kell megtervezni, hogy a két fél között a lehető legkevesebb adatátvitelt kelljen végrehajtani.

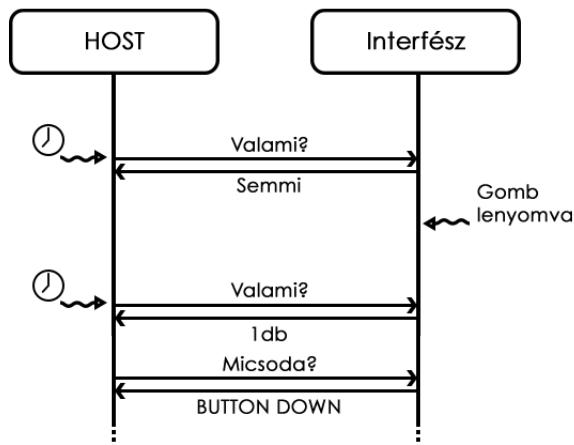
4.2. AVR és SHARC - közös nevező

Mind a hoszt egység SHARC jelprocesszora, mind az interfész egység AVR mikrovezérője tartalmaz hardveres SPI egységet. Ahhoz, hogy a kommunikáció megfelelően működjön közük, azonos módon kell felkonfigurálni minden a SHARC jelprocesszor által támogatott képességei miatt nem lehet kihasználni minden a SHARC jelprocesszor által támogatott gyorsítási lehetőséget, de a rendszer kevésbé időkritikus mivolta ezt nem is igényli.

A kommunikációt a hoszt egység vezéri minden körülmények között. Ő kérdezi le az eseményeket periodikusan az interfész eszközről, valamint ő küldi át a megjelenítendő adatokat is. Az ütemezés állítható hosszúságú, de minden a DSP rendszer mintavételi idejének a többszöröse. A lekérdezés gyakoriságának az AVR számítási kapacitása adja a felső, a rendszer reakcióideje pedig az alsó határát.

4.3. Protokoll felépítése

A protokoll kérdezz-felelek elven működik. A hoszt egység ütemezetten megkérdezi az interfészről, hogy történt-e valami esemény. Ha történt, a hoszt egyesével lekérdezi azokat, ha nem, várja a következő lekérdezési időpontot. (4.1. ábra)



4.1. ábra. SPI csomagok kódolása

- a hoszt üzenetet küld az eszköznek, amivel jelzi, hogy le fogja kérdezni az eseményeket
- az üzenet átvitele után a hoszt tudja, hogy mennyi lekérdezést kell végeznie
- ha történtek események a legutóbbi lekérdezés óta, akkor a hoszt egyesével lekérdezi őket

Az SPI busz full duplex átvitelű, és ezt a protokoll ki is használja. Az interfész egység az SPI hardvere kimeneti regiszterébe mindenig az aktuálisan ki nem olvasott események számát tölti, így a hoszt egység a lekérdezési üzenet átvitele után azonnal tudja, hogy hány esemény történt a legutóbbi lekérdezés óta, és ez alapján ütemezi a további lekérdezéseket. Ebből következik, hogy a hoszt mindenkor csak annyi lekérdezést végez, amennyi minimálisan szükséges.

A lekérdezések ütemezése a DSP rendszer mintavételi frekvenciája szerint van ütemezve, azonban, hogy elkerüljük a felesleges üres lekérdezéseket, a jelenlegi konfigurációban 48 kHz-en 26-szor lassabban, nagyjából 1.8 kHz gyakorisággal kérdezzük le az eseményeket.

Mivel a interfész eszköz képes adatok megjelenítésére, ezért a protokoll támogatja adatok küldését is. Ekkor az átvitel azonos a csak lekérdezést tartalmazó átvitelkel, azzal az eltéréssel, hogy a hoszt a lekérdezések alatt adatokat küld, az interfész eszköz pedig fogadja azokat:

- a hoszt üzenetet küld az eszköznek, ami tartalmazza az küldendő adat típusát
- az üzenet átvitele után a hoszt tudja, hogy kell-e az adatok átléptetése közben fogadnia is eseményeket
- az eszköz miután megkapta az átvitelt jelző üzenetet, felkészül az adatok fogadására
- a hoszt sorban kiküldi az adatokat, és ha szükséges, fogadja az eseményeket

Az adatküldések minden fix számú ciklusból áll, aminek a számát az adott típususa határozza meg. Ha az adatátviteli ciklusok száma kevesebb, mint a kiolvasásra váró események száma, akkor a maradék esemény egy következő alkalommal lesz kiolvasva. Ez egy olyan egyszerűsítés, amit a felhasználó nem vesz észre, de az SPI kommunikációt kezelő modul állapotgépét jelentősen leegyszerűsíti.

Az SPI protokoll a különböző átviteli módoknak különböző átviteli gyakoriságot enged. Ennek az oka az SPI busz túlterhelésének az elkerülése. A legtöbb időt igénylő átvitel az LCD kijelző adatainak az átvitele. Az SPI-t ütemezője ezért ennek a típusú átvitelnek adja a legkevesebb buszidőt. Ez azért is jó, mert maga az LCD egység a többi egységhez képest jóval lassabban képes frissíteni a tartalmát.

A LED-es oszlopok átvitele sokkal rövidebb idő alatt megvan, és az LED-es megjelenítés is sokkal gyorsabb, ezért az LCD buszidejénél jelentősen többet kap az LED-adatok átvitele.

Az események lekérdezésén kívül a DIP kapcsolók állását is periodikusan lekérdezi a hoszt egység. A jelenlegi konfigurációnál minden 20. lekérdezés egy DIP kapcsoló lekérdezés.

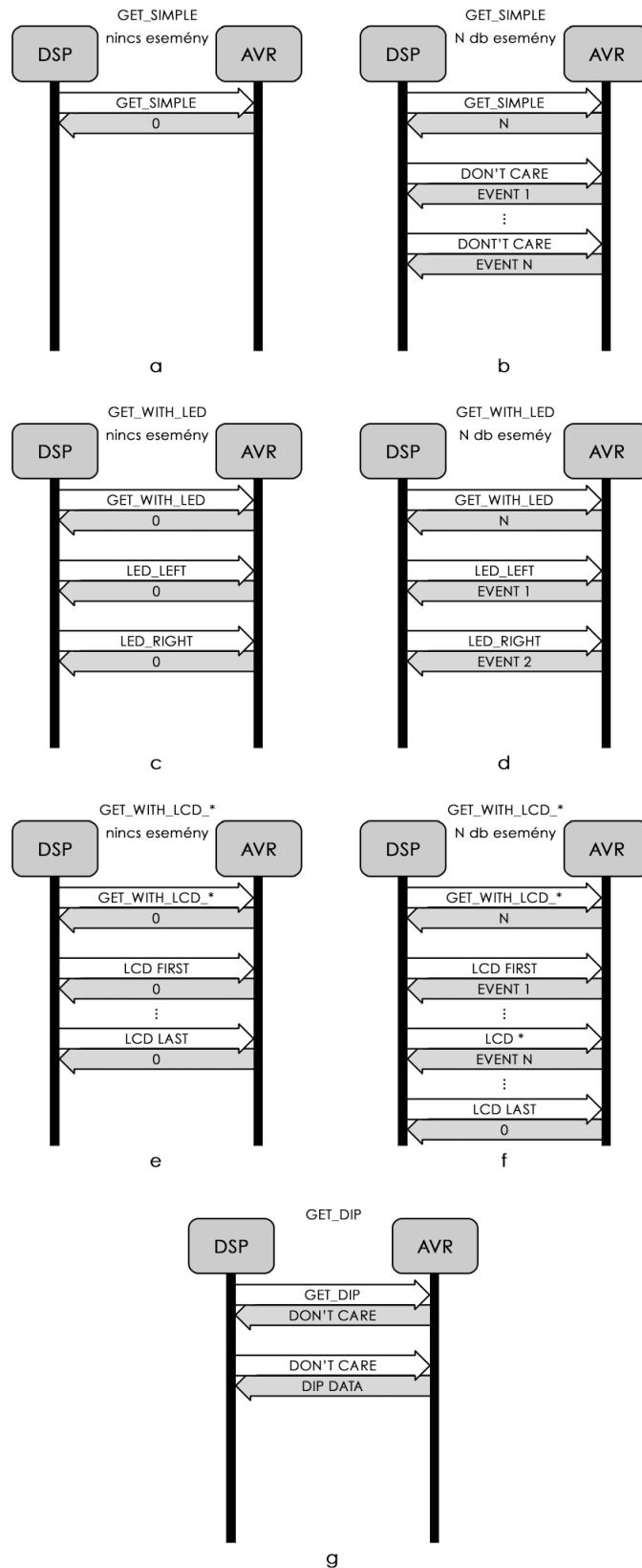
4.3.1. Típusok

Az előző részből kiderült, hogy többféle átviteli típus lehetséges az SPI kommunikációban. Ebben az alfejezetben részletes bemutatásra kerülnek az egyes típusok.

A host egység minden kommunikációs ciklus elején egy azonosítót küld az interfész eszköznek, amivel jelzi a szándékát. Ezek az azonosítók a következők lehetnek:

GET_SIMPLE	Események lekérdezése.
GET_WITH_LED	Események lekérdezése LED adatok átvitelével.
GET_WITH_LCD_TOP	Események lekérdezése a felső LCD sor adataival.
GET_WITH_LCD_BOTTOM	Események lekérdezése az alsó LCD sor adataival.
GET_DIP_STATUS	DIP kapcsolók állásának lekérdezése.

A 4.2. ábrán látható az összes lehetséges átviteli üzemmódnak megfelelő protokoll időbeli lefutása.



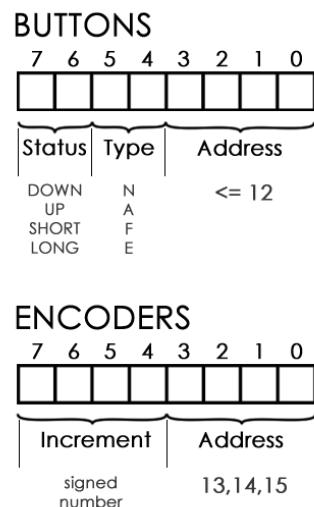
4.2. ábra. SPI átviteli típusok

A 4.2. ábrán vízszintesen egymás mellett vannak az azonos lekérdezési típusok. A bal oldalon nem történt esemény, amit le kéne kérdezni, a jobb oldalon igen. Az előzőekben tárgyalt viselkedés végigkövethető az ábrákon.

Az *e* és az *f* ábrák tetszőlegesen alsó vagy felső LCD frissítésre igazak.

4.3.2. Csomagolás

A fejezet elején említettük, hogy az interfész egység központjaként funkcionáló AVR korlátozásai miatt a küldendő adatok méretére oda kell figyelni. A protokoll tervezésénél kiiderült, hogy egy 8 bites csomagba elfér az összes bekövetkezni képes esemény. A következő csomagolási eljárást alkalmazzuk:



4.3. ábra. SPI csomagok kódolása

A 8 bitet kisebb csoportokra osztjuk. Az első két bit az esemény típusát jelöli (up, down, long, short), a második két bit az eseményhez tartozó nyomógomb csoportját (enkóder, funkció, navigációs, numerikus), majd a maradék 4 bit a nyomógomb csoporton belüli indexét jelöli.

Ez a kódolás lefedi az összes nyomógomb összes eseményét. Viszont az eseménykezelő rendszer az enkóderek forgatását is feldolgozza. Kihasználjuk, hogy a 3. bitcsoport értéke maximum 12 lehet, mivel a legtöbb gombot tartalmazó numerikus csoport legnagyobb indexű gombja az $N12$. Marad még három kombináció 4 biten, amit az enkóderek forgatásának jelzésére használunk. Ekkor az előzőektől eltérően a felső 4 bit nem esemény típusát és csoportját, hanem a legutóbbi kiolvasás óta történt tekerések számát ábrázolja egy 4 bites előjeles számmal.

A DIP kapcsolólók kódolásáról még nem esett szó. Ezek nem tartoznak gyors reagálású eseménykezelő rendszer hatáskörébe. A hoszt egység ritkább időközönként lekérdezi az állapototukat, ami egy egy bájtos adat átvitelét jelenti, amiben a kapcsolók állásának megfelelő kombináció szerepel.

5. fejezet

DSPController API v1.0

A következő fejezetben ismertetésre kerül a DSPController API, ami lehetőséget biztosít a programozónak az interfész eszköz programkód szintű felhasználására. Először az API által definiált típusokat és eseményeket ismertetjük, majd az API magját képező függvények tárgyalása következik, végül pedig a kiegészítő függvénycsomagok használatáról esik szó.

5.1. Típusdefiníciók

Az API négy típust definiál, amelyek egyszerűsítik és egyértelművé teszik a programkódot.

Event	<code>typedef unsigned char Event</code>	Események kezelésére
DIP	<code>typedef unsigned char DIP</code>	DIP kapcsolók kezelésére
Encoder	<code>typedef signed short Encoder</code>	Enkóderek értékének kezelésére
Led	<code>typedef unsigned char Led</code>	Led oszlopok kezelésére

Programozás során célszerű ezen típusdefiníciók használata, mivel velük elkerülhető a nem egységes típusok használata során felmerülő konverziós adatvesztés.

5.2. Események

A DSPController rendszerben az interfész eszközön található gombok mindenkorához definiálva van négy eseménytípus. Ezeket az eseményeket képes a rendszer megkülönböztetni.

DOWN	Lenyomás pillanatában generálódó esemény.
UP	Felengedés pillanatában generálódó esemény.
SHORT	Felengedés pillanatában generálódó esemény, amennyiben a nyomvattartás ideje nem lépte át a hosszú nyomás határidejét.
LONG	A hosszú nyomás határidejének átlépésekor generálódó esemény. Kizárt vagy kapcsolatban van a SHORT eseménnyel.

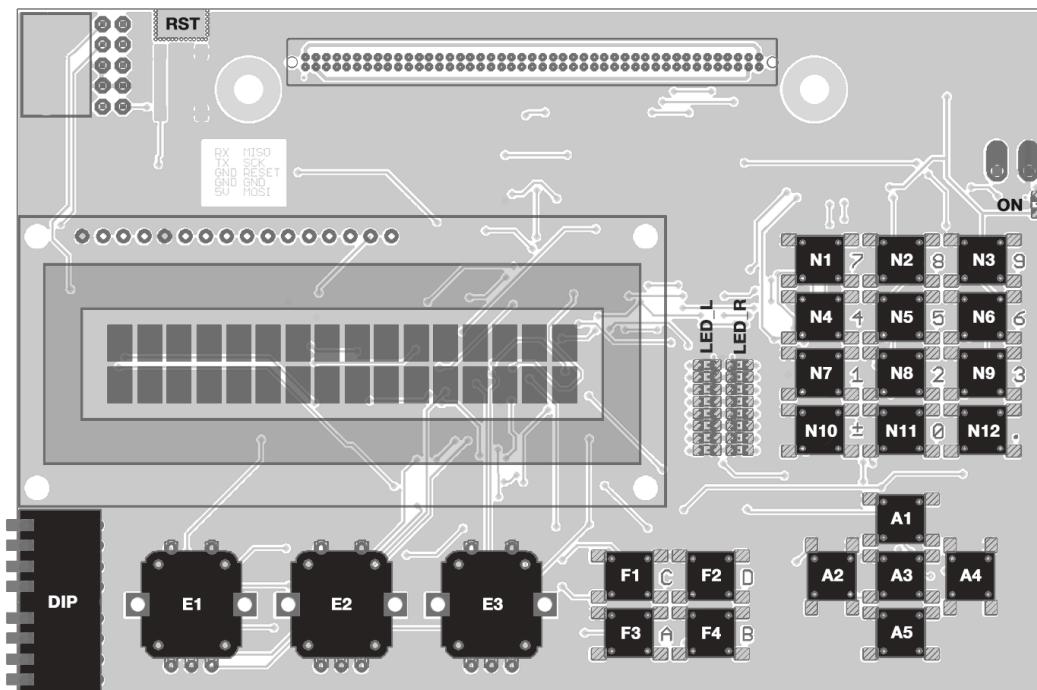
A szoftveres API minden eseményhez hozzárendel egy számot, amivel azokat kezelni

tudja. A programozónak nem kell fejből tudnia az adott eseményhez tartozó számot. A DSPController API biztosít egy szimbólumcsomagot, ami a logikus elnevezési konvenció alapján, könnyen azonosíthatóvá teszi az egyes eseményeket.

Események elnevezési konvenciója			
DSPC_	EVENT_	XY_	DOWN UP SHORT LONG
DSPController tipikus szimbólum kezdése	Esemény szimbólum jelzése	X: gomb csoportja, Y: gomb csoporton belüli száma	Esemény típusa

A fenti logika és a felülnézei rajz alapján (5.1. ábra) könnyen előállítható bármelyik nyomógomb bármely eseménye. Például a bal szélső enkóder nyomógomb felengedésének a szimbóluma: DSPC_EVENT_E1_UP

Ezt közvetlenül fel lehet használni a programban komparálásra.



5.1. ábra. DSPController eszköz felülnézet

A DIP kapcsolókra nincsenek definiálva események, mivel csak két állapotuknak van értelme (bekapcsolt vagy kikapcsolt). A kapcsolók lekérdezésére más eszközök állnak rendelkezésre.

Hasonlóan a DIP kapcsolókhöz, az enkóderekhez sincs külön esemény definiálva (az enkóderek nyomógombja ez alól kivétel). Egy dedikált mechanizmuson keresztül érhetők el az aktuális enkóder értékeit. Ezt és a DIP kapcsolók lekérdezését a későbbiekbén részletesen tárgyaljuk.

5.3. Core API függvények

A DSPController API magját 11 függvény képezi.

void DSPController_init_default(void)	5.3.1
void DSPController_init_init(void)	5.3.2
void DSPController_tick(void)	5.3.3
void DSPController_flush(void)	5.3.4
void DSPController_led(Led leftLed, Led rightLed)	5.3.5
void DSPController_lcd_top(const char* format, ...)	5.3.6
void DSPController_lcd_bottom(const char* format, ...)	5.3.7
void DSPController_lcd(char line, const char* format, ...)	5.3.8
DIP DSPController_get_dip(void)	5.3.9
Event DSPController_get_event(void)	5.3.10
Encoder DSPController_get_encoder(char encoder)	5.3.11

5.3.1. void DSPController_init_default(void)

Függvény: void DSPController_init_default(void)

Előfeltétel: Nincs

Bemenet: Nincs

Kimenet: Nincs

Mellékhatás: Nincs

A függvény hívása után a rendszer készen áll a működésre az alapértelmezett beállítások szerint. A hívást célszerű az INTERFACE.c - initInterface() függvényében egyszer elvégezni.

Alapértelmezett beállítások:

- **48 kHz** mintavételi frekvencia
- Enkóder sebesség figyelembevétel **bekapcsolva**

5.1. példakód. Rendszer inicializálása alapértelmezett beállításokra

```
1 // INTERFACE.c fájl
2 void initInterface(void) {
3
4     DSPController_init_default();
5
6 }
```

5.3.2. void DSPController_init(int code)

Függvény: void DSPController_init(int code)

Előfeltétel: Nincs

Bemenet: Rendszer beállítások kódolva

Kimenet: Nincs

Mellékhatás: Nincs

A függvény hívása után a rendszer készen áll a működésre a megadott beállítások szerint. A hívást célszerű az INTERFACE.c - initInterface() függvényében egyszer elvégezni.

Fontos, hogy ezek a beállítások csak a DSPController rendszerre érvényesek, azaz az itt beállított mintavételi frekvencia nem fogja megváltoztatni a DSP rendszer mintavételi frekvenciáját, csupán a DSPController drivernek adja meg az időzítéseket. (A DSP rendszer mintavételi frekvenciáját a CODEC felkonfigurálásánál lehet megváltoztatni, ami nem esik a DSPController API hatáskörébe.)

Két paraméter beállítására van lehetőség

- **Mintavételi frekvencia** - A beállított értéknek megfelelően időzíti a driver a kommunikációt.
 - FS_48KHZ Driver 48 kHz-en üzemel.
 - FS_96KHZ Driver 96 kHz-en üzemel.
- **Enkóder sebesség figyelembevétel** - Bekapcsolt állapotban az enkóderek gyors tekerésénél a tekert érték háromszorosát adják vissza. Nagy tartományok átfogásánál gyorsítja az interfész használatát.
 - ENCODER_VELOCITY_ON Gyorsulásfigyelés bekapcsolva.
 - ENCODER_VELOCITY_OFF Gyorsulásfigyelés kikapcsolva.

A beállítási lehetőségek szimbólumok, amelyek számokat kódolnak. Egyszerre több beállítás is megadható, ekkor az egyes beállításokat logikai vagy kapcsolatba kell hozni egymással. Ha nem adunk meg valamely beállításra egy konkrét értéket, akkor az alapértelmezett beállítást állítja be a függvény az adott paraméterre. (lásd: 5.3.1)

5.2. példakód. Rendszer manuális inicializálása

```

1 // Rendszer inicializálása az INTERFACE.c fájlban.
2 // Beállítások:
3 // 48 kHz mintavételi frekvencia
4 // Enkóder gyorsulás figyelése bekapcsolva
5 void initInterface(void) {
6     DSPController_init( FS_48KHZ | ENCODER_VELOCITY_ON );
7 }
```

5.3.3. void DSPController_tick(void)

Függvény:	void DSPController_tick(void)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Nincs
Kimenet:	Nincs
Mellékhatás:	Nincs

DSPController driver ütemező függvénye. Periodikus hívást igényel minden ADC megszakításnál. Feladata kezeli a driver állapotgépet, ütemezi a SPI kommunikációt, és adminisztrálja a bejövő eseményeket.

Elengedhetetlen fontosságú feladatokat lát el, periodikus hívása nélkül működés képtelen a rendszer.

Ez az **egyetlen** API függvény, ami **időkritikus szakaszból** hívható. minden más API függvény szigorúan csak alacsony prioritású szakaszból hívható.

5.3.4. void DSPController_flush(void)

Függvény:	void DSPController_flush(void)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Nincs
Kimenet:	Nincs
Mellékhatás:	Minden fel nem dolgozott esemény elveszik

Hívásával törlődik minden eseménytáróló bufferben lévő esemény, valamint az enkóderek értéke is alaphelyzetbe áll.

Használata interfész állapotok közötti váltásnál javasolt, mikor az aktuális állapot nem kéri le az összes eseményt vagy enkóder értéket, de a következő állapot igen, és a le nem kért események az állapotváltás után azonnal feldolgozásra kerülnek. Ekkor azonban a felhasználó által követhetetlen sebességgel nem kívánt állapotba viszik a rendszert.

(egyszerűsítve: állapotváltásnál látszólag nem oda lép a rendszer, ahova kellene neki)

5.3. példakód. Állapotváltás előtti eseménytörles

```
1 // Az aktuális állapotban nincsenek használva az enkóderek, de a
2 // felhasználónak nem lehet megtiltani a tekerésüköt.
3 // Lekérdezetlenül a tekert értékek kummulálódnak.
4 if(e == DSPC_EVENT_A3_SHORT) {
5     // A következő állapot használja az enkódereket. Állapotváltás
6     // után azonnal feldolgozná a kummulált tekeréseket.
7     interface_state = STATE_WITH_ENCODER_USAGE;
8     DSPController_flush(); // Ezért állapotváltáskor minden törliünk
9 }
```

5.3.5. void DSPController_led(Led leftLed, Led rightLed)

Függvény:	void DSPController_led(Led leftLed, Led rightLed)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Új kivezérlésjelző értékek
Kimenet:	Nincs
Mellékhatás:	Nincs

Új adat küldése a kivezérlésjelző led oszlopoknak. Hívásakor a paraméterként megadott bal és jobb 8 bites adat bekerül a kimeneti bufferbe, ahonnan a driver ütemezője elküldi az eszközre, ami megjeleníti azt.

5.4. példakód. Led oszlopok frissítése

```
1 // Ledes oszlopok frissítése adott értékekre
2 Led left = 0x55;
3 Led right = 0xaa;
4
5 DSPController_led(left,right);
```

5.3.6. void DSPController_lcd_top(const char* format, ...)

Függvény:	void DSPController_lcd_top(const char* format, ...)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Felső LCD sor formázott karaktersorozata
Kimenet:	Nincs
Mellékhatás:	Nincs

Az LCD kijelző felső sorát frissíti a megadott formázás szerinti karaktersorozattal. Használata ekvivalens a könyvtári printf() függvény használatával, azzal a különbséggel, hogy lebegőpontos számok kijelzésére alkalmatlan. Ehhez egy később tárgyalt kiegészítő függvénycsomagot kell használni. (DSPC_FTS: 5.4.2.1.1)

Egy sorban összesen 16 karakter fér el. Ha több karaktert adunk meg, a felesleges karakterek nem kerülnek átvitelre. Ha kevesebb karaktert adunk meg, a hiányzó részt a függvény szóközökkel tölti fel. Egy LCD sor frissítésnél mindenkor minden sor frissül, nem maradnak meg a korábbi frissítéskor átvitt karakterek.

5.5. példakód. LCD felső sorának frissítése

```
1 // LCD kijelző felső sorának frissítése.
2 int number = 42;
3 DSPController_lcd_top("The number is %d",number);
4 // Az eredmény: [The number is 42]. Pont 16 karakter.
```

5.3.7. void DSPController_lcd_bottom(const char* format, ...)

Függvény:	void DSPController_lcd_top(const char* format, ...)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Alsó LCD sor formázott karaktersorozata
Kimenet:	Nincs
Mellékhatás:	Nincs

Használata és működése azonos a felső sort frissítő függvényel, azzal a különbséggel, hogy ez a függvény az alsó LCD sort frissíti.

5.3.8. void DSPController_lcd(char line, const char* format, ...)

Függvény:	void DSPController_lcd(char line, const char* format, ...)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Adott LCD sor formázott karaktersorozata
Kimenet:	Nincs
Mellékhatás:	Nincs

Az előző két függvényel ekvivalens LCD kezelő függvény. Programozás során sok helyen hasznos lehet, ha egymás alatt látjuk az alsó és felső LCD sorra küldendő formázott szöveget. Ezt az előző alsó és felső sorra specifikált függvénynél csak felesleges szóközök beiktatásával érhetjük el. Ennek a kényelmetlenségnek a feloldására létezik az LCD kezelő függvényeknek a harmadik változata, ami a formázott karaktersorozaton kívül a frissítendő sor számát is kéri paraméterként.

5.6. példakód. LCD felső sorának frissítése

```
1 // A két LCD frissítés azonos, a forráskód áttekinthetősége azonban
2 // jobb a második esetben.
3 DSPController_lcd_top("__Fc__Q__Mod__");
4 DSPController_lcd_bottom("_%4d_%2d_sine_",fc,q);
5
6 DSPController_lcd(0, "__Fc__Q__Mod__");
7 DSPController_lcd(1, "_%4d_%2d_sine_",fc,q);
8 // A LCD karaktersorozatai egymás alá esnek, úgy, ahogy a kijelzőn
9 // megjelennének.
10 // A pédkódban az aláhúzások szóközöket jelölnek
```

5.3.9. DIP DSPController_get_dip(void)

Függvény:	DIP DSPController_get_dip(void)
Előfeltétel:	Inicializált DSPController driver, és az inicializálás után legalább 0.54 ms várakozási idő, amíg az első DIP kiolvasás megtörténik alacsony szinten.
Bemenet:	Nincs
Kimenet:	Utoljára kiolvasott DIP kapcsoló állása bináris kódolásban
Mellékhatás:	Nincs

A DSPController driver periodikusan lekérdezi az aktuális DIP kapcsoló állásokat az eszközről, és elérhetővé teszi a kiolvasott értéket. A függvény ezt az értéket adja vissza egy 8 bites szám formájában, ahol az magas logikai szintű bit jelenti a kapcsoló zárt állapotát. Az LSB bit számít a legalsó kapcsolónak, ha az interfész egységre szemből nézünk rá.

5.7. példakód. DIP kapcsolók kezelése

```
1 // DIP kapcsolók lekérdezése
2 DIP dip = DSPController_get_dip();
3
4 // Legfelső kapcsoló bekapcsolt állapotban van-e?
5 if ((dip & 0x80) != 0) {
6     // Legfelső kapcsoló bekapcsolva
7 } else {
8     // Legfelső kapcsoló kikapcsolva
9 }
```

5.3.10. Event DSPController_get_event(void)

Függvény:	Event DSPController_get_event(void)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Nincs
Kimenet:	Eseménybuffer legfelső eleme
Mellékhatás:	Nincs

Az egyik legtöbbet használt API függvény. Hívása során visszaadja a eseménybuffer legfelső elemét. Ha az eseménybuffer üres, akkor a DSPC_EVENT NOTHING eseménnyel tér vissza.

Hívásnál a visszaadott esemény törlődik az eseménybufferból. Többszörös teszteléshez ezért célszerű az eseményt letárolni egy változóba, majd erre a változóra komparálni. Ha minden komparálásnál az API függvényt használnánk **hibásan**, akkor bizonyos ágakba sose jutna el a vezérlés, hiszen a hozzá szükséges eseményt egy előző ágban "felhasználtuk".

Az eseménybuffer mérete állítható ugyan, de érdemes minden esetben lekérdezni az eseményeket, még akkor is, ha nem használjuk fel őket. Ha csak akkor kezdenénk el lekérdezni az eseményeket, mikor a programkód megkezdi a tényleges felhasználásukat, akkor az eddig felhalmozódott események megzavarhatják az vezérlés futását. Ha mégsem kérdezzük le az eseményeket folyamatosan, akkor érdemes kiüríteni az eseménybuffert mielőtt megkezdjük az eseményekre való komparálást. (lásd: 6.4.2.2.3)

5.8. példakód. Események lekérése és feldogozása

```

1 // Interfész update függvényén belül
2 // Új esemény átmeneti tárolása
3 Event e = DSPController_get_event();
4
5 // Esemény tesztelése és szétválasztása
6 switch(e) {
7     case DSPC_EVENT_NOTHING:
8         // Nem történt semmi.
9         break;
10    case DSPC_EVENT_F1_SHORT:
11        // F1 gombnál rövid nyomás volt
12        break;
13    case DSPC_EVENT_F2_UP;
14        // F2 gombot felengedték
15        break;
16    ...
17    default:
18        break;
19 }
```

5.3.11. Encoder DSPController_get_encoder(char encoder)

Függvény:	Encoder DSPController_get_encoder (char encoder)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Lekérdezendő enkóder azonosító száma
Kimenet:	Adott enkóder legutóbbi kiolvasás óta változott értéke.
Mellékhatás:	Kiolvasás után nullázódik az adott enkóder számlálója

A függvény segítségével kiolvasható a kívánt enkóder legutóbbi kiolvasás óta változott tekerési értéke. Ha nem történt semmi változás, a függvény nullát ad vissza, egyébként egy előjeles egész számot. Kiolvasás után nullázódik az adott enkóder értéke, következő kiolvasásnál, ha nem tekerték el, nullát fog visszaadni a függvény.

Az óramutató járásával egyező irányú tekerés növeli, ellenkező irányú tekerés csökkenti az enkóder értékét. Felkonfigurálástól (5.3.2) és tekerési sebességtől függően egy kattanásra többet is léphet az enkóder értéke.

5.9. példakód. Enkóder kezelése

```

1 // Globális változó az enkóder értékének a tárolására
2 // Interfész frissítő függvényén kívül deklarálva
3 Encoder e1 = 0;
4 ...
5
6 // Interfész frissítő függvényén belül: Enkóder értékének frissítése
7 e1 += DSPController_encoder(1);

```

5.4. Kiegészítő függvénycsomagok

A DSPController API v1.0 a core függvényeken kívül tartalmaz még további két függvénycsomagot, amelyek a programozó dolgát hivatottak megkönnyteni.

5.4.1. Assembler

A DSPController Assembler csomag jelentős segítséget nyújt a programozónak, ha az alkalmazás a numerikus billentyűzetet numerikus billentyűzetként használja. Ekkor a gombok lenyomási szekvenciából kellene összeállítani a felhasználó által éppen bevinni kívánt számot. Az Assembler csomag pontosan ezt a feladatot látja el.

Képességek:

- Beviteli szekvenciából lebegőpontos szám előállítása a begépelés során folyamatosan.
- Automatikus beviteli szekvencia vége észlelés.
- Utoljára begépelt szám eltárolása.
- 7 jegyű egészrész és 31-egészrész jegyű törtrész kezelése.
- Alacsony szinten integrált, magas szinten kezelhető.
- Tetszőleges projecthez fordítás.

5.4.1.1. Assembler használata

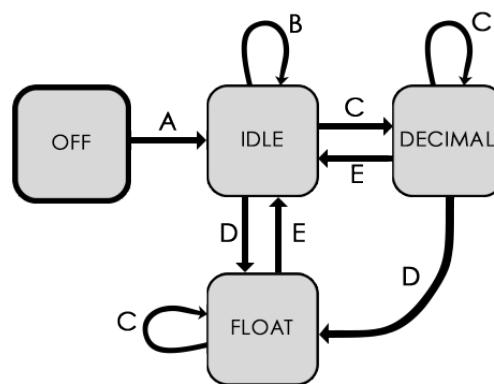
Az Assembler modul aktiválás után (5.4.1.2.2) várakozó állapotba kerül, és figyeli az eseményeket. Ha a felhasználó egy numerikus gombot nyomott le (N1-N12), azonnal feldolgozza, és elérhetővé teszi az aktuális konvertált számot a `DSPController_assembler_get_current()` (5.4.1.2.4) függvényen keresztül. A felhasználó minden új numerikus gombnyomására megtörténik a konverzió, és lekérhető a frissen konvertált szám. Az Assembler nem "használja el" a numerikus billentyűzet lenyomási eseményét, az elérhető marad a DSPController API esemény lekérdező `DSPController_get_event()` (5.3.10) függvényén keresztül.

Ez a folyamat addig tart, amíg a felhasználó be nem fejezi a szám begépelését a numerikus billentyűzetén. Bármely nem numerikus gomb lenyomásakor az Assembler befejezi a szám konvertálását,

elmenti a konvertált számot, ami a `DSPController_assembler_get_last()` (5.4.1.2.3) függvényel érhető el, bebillent egy flag-et, ami a `DSPController_assembler_is_new_number()` (5.4.1.2.6) függvényen keresztül érhető el, valamint alaphelyzetbe állítja a konvertáló mechanizmúsát, felkészülve az új beviteli szekvenciára.

Abban a különleges helyzetben, mikor a felhasználó a szám bevitelének a végét nem egy másik típusú gomb lenyomásához akarja kötni, hanem programozottan egy bizonyos hosszúságú vagy értékű szám begépeléséhez, enkóder forgatáshoz, vagy DIP kapcsoló állapotváltásához, akkor programból kényszerítheti az Assemblert, hogy fejezze be a szám konvertálását a `DSPController_assembler_finish_number()` (5.4.1.2.5) függvényel. Ekkor a DSPController Assembler teljesen azonosan jár el, mint az automatikus befejezésnél. Elmenti a konvertált számot, flag-et billent, és felkészül az új beviteli szekvenciára.

Az Assembler állapotgépe a 5.2. ábrán látható.



5.2. ábra. *DSPController Assembler állapotgépe*

Az állapotgép állapotátmeneti feltételei az alábbi táblázatban láthatóak:

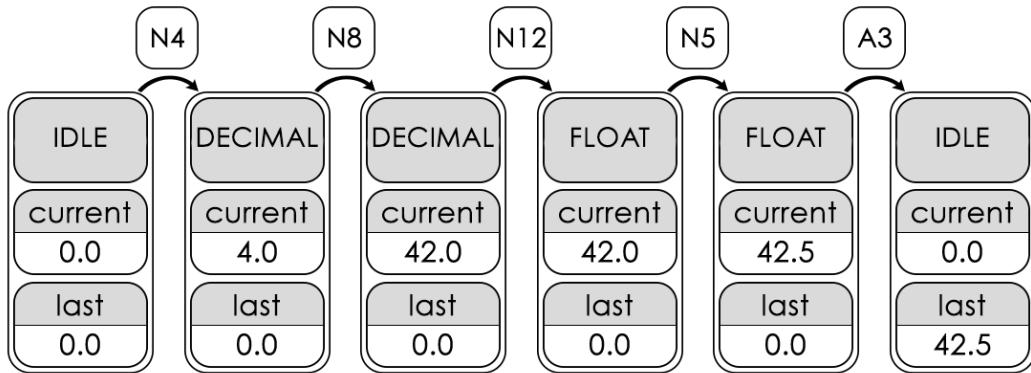
- A** `DSPController_assembler_engage()` függvény hívása.
- B** Bármelyik nyomógomb lenyomása a numerikus nyomógombokon kívül.
- C** Bármelyik numerikus nyomógomb lenyomása a decimális ponton kívül (*N12*).
- D** Decimális pont lenyomása (*N12*).
- E** *B* feltétel vagy a `DSPController_assembler_finish_number()` függvény hívása.

Az DSPController Assembler működésének alaposabb megismeréséhez az 5.3. ábra bemutat egy beviteli szekvenciát, ami közvetlenül inicializálás után kezdődik. Az ábrán látható, hogyan változnak a `DSPController_assembler_get_current()` és a `DSPController_assembler_get_last()` függvények által visszadott értékek.

A felhasználó a 42.5 értéket kívánja bevinni az eszközzel, ehhez sorban lenyomja a megfelelő gombokat: 4, 2, ., 5, majd a középső navigációs gombot (feltételezzük, hogy ez a gomb véglegesíti a bevitelt az adott alkalmazásban).

A bevitel végére az Assembler előállította a felhasználó által bevinni kívánt értéket, ami elérhető tőle. A szekvencia befejeztről a `DSPController_assembler_is_new_number()` flag-lekérdezéssel bizonyosodhatunk meg.

Amennyiben a bevitel során folyamatosan frissíteni szeretnénk az LCD kijelzőn a bevitt számot, úgy a `DSPController_assembler_get_current()` függvényel beviteli szekvencia közben lekérhető az aktuális számérték.



5.3. ábra. DSPController Assembler példa szekvencia

5.10. példakód. Assembler tipikus használata

```

1 // Az Assembler modul előzőleg aktiválásra került
2 // INTERFACE.c update függvényén belül
3 ...
4
5 // Célszerű lekérdezni az eseményeket akkor is, ha nem használjuk
6 // fel őket, hogy elkerüljük az eseménybuffer telítődését.
7 Event e = DSPController_get_event();
8
9 // A példakód minden numerikus gombnyomásnál kiírja a frissen
10 // konvertált számot a felső LCD sorba, majd a szekvencia végén
11 // a kész számot az alsó LCD sorba. A kódban a később
12 // tárgyalandó float to string konverter is használva van.
13 float temp_float = DSPController_assembler_get_current();
14 DSPController_lcd_top("New: %s", DSPC_FTS(temp_float, 6));
15
16 if (DSPController_assembler_is_new_number()) {
17     float last_float = DSPController_assembler_get_last();
18     DSPController_lcd_bottom("Last: %s", DSPC_FTS(last_float, 6));
19 }
```

5.4.1.2. Assembler API

Az Assembler API öt függvényből áll, amiből egy csak nagyon kivételes esetben használatos.

void DSPController_assembler_engage(void)	5.4.1.2.2
DSPC_CONV_TYPE DSPController_assembler_get_last(void)	5.4.1.2.3
DSPC_CONV_TYPE DSPController_assembler_get_current(void)	5.4.1.2.4
DSPC_CONV_TYPE DSPController_assembler_finish_number(void)	5.4.1.2.5
unsigned char DSPController_assembler_is_new_number(void)	5.4.1.2.6

5.4.1.2.1. DSPC_CONV_TYPE szimbólum

Az API definiál a DSPC_CONV_TYPE szimbólumot, ami az összerakott lebegőpontos számok típusát jelöli. Ezzel a típussal térnek vissza a konvertáló függvények. Az alapértelmezett típus a float, ami módosítható a DSPController.h header fájlban double típusra is, ha a szimpla pontosság nem ad elég felbontást.

5.4.1.2.2. void DSPController_assembler_engage(void)

Függvény:	void DSPController_assembler_engage(void)
Előfeltétel:	Inicializált DSPController driver
Bemenet:	Nincs
Kimenet:	Nincs
Mellékhatás:	Nincs

A függvény aktiválja az Assembler modult. A hívást célszerű az INTERFACE.c – initInterface() függvényében, a DSPController driver inicializálása után egyszer elvégezni.

5.11. példakód. Assembler aktiválása

```

1 // Rendszer és Assembler inicializálása az INTERFACE.c fájlban
2 void initInterface(void) {
3     DSPController_init( FS_48KHZ | ENCODER_VELOCITY_ON );
4     DSPController_assembler_engage();
5 }
```

5.4.1.2.3. DSPC_CONV_TYPE DSPController_assembler_get_last(void)

Függvény:	DSPC_CONV_TYPE DSPController_assembler_get_last(void)
Előfeltétel:	Inicializált DSPController driver és aktív Assembler modul
Bemenet:	Nincs
Kimenet:	Lebegőpontos szám, ami a befejezett beviteli szekvencia eredménye
Mellékhatás:	Nincs

Mikor a felhasználó befejezi a beviteli szekvenciát (megnyom egy nem numerikus gombot vagy a szoftver meghívja a DSPController_assembler_finish_number() (5.4.1.2.5) függvényt), a kész konvertált számot elmenti az Assembler. A mentett szám mindenkorán elérhető lesz ezen a függvényen keresztül, amíg egy újabb szám konvertálása be nem fejeződik.

5.4.1.2.4. DSPC_CONV_TYPE DSPController_assembler_get_current(void)

Függvény:	DSPC_CONV_TYPE DSPController_assembler_get_current (void)
Előfeltétel:	Inicializált DSPController driver és aktív Assembler modul
Bemenet:	Nincs
Kimenet:	Beviteli szekvencia során a legfrissebb konvertált szám
Mellékhatás:	Nincs

A függvény visszaadja a felhasználó által begépelet legutóbbi számot. Ha a beviteli szekvencia végetéért (a felhasználó egy nem numerikus gombot nyomott meg), a függvény 0.0f értéket ad vissza mindaddig, amíg a felhasználó bele nem kezd egy új beviteli szekvenciába, azaz le nem nyom egy numerikus gombot.

5.4.1.2.5. DSPC_CONV_TYPE DSPController_assembler_finish_number(void)

Függvény:	DSPC_CONV_TYPE DSPController_assembler_finish_number (void)
Előfeltétel:	Inicializált DSPController driver és aktív Assembler modul
Bemenet:	Nincs
Kimenet:	Lebegőpontos szám, ami a befejezett beviteli szekvencia eredménye
Mellékhatás:	Nincs

Szoftveresen kényszerített befejezés a konvertálási szekvenciához. Ha a programozónak nem megfelelő az Assembler automatikus befejezése, és a konverzió végét más eseményhez szeretné kötni, mint egy nem numerikus billentyű lenyomása, akkor használhatja ezt a függvényt a megszakításra. A függvény hívásakor vissza is tér a befejezett és lementett konvertált számmal.

5.4.1.2.6. unsigned char DSPController_assembler_is_new_number(void)

Függvény:	unsigned char DSPController_assembler_is_new_number (void)
Előfeltétel:	Inicializált DSPController driver és aktív Assembler modul
Bemenet:	Nincs
Kimenet:	Beviteli konverzió végét jelző flag
Mellékhatás:	Nincs

Flag lekérdező függvény ami nem nulla értékkel tér vissza, ha a konverzió automatikusan vagy mesterségesen vége lett.

Kiolvasásnál törlődik a flag, ismételt kiolvasásra már nulla értékkel tér vissza a függvény.

5.4.2. Float to String Converter

A Float to String Converter a beágyazott rendszerekben megsokott könyvtári stdio csomag tipikus hiányosságát hivatott pótolni. Bevett eljárás, hogy beágyazott rendszerekben az stdio csomagba nem fordítják bele a lebegőpontos számok kezelését, mivel az ehhez szükséges általános kód túl sok helyet foglalna. A DSPController szoftveres környezetében azonban szükség van lebegőpontos számok kijelzésére is. Erre az igényre készült a Float to String Converter modul, ami egy leszűkített képességű, de gyors és kis méretű float to string konverter.

5.4.2.1. Konvertáló függvény

A programcsomag egyetlen függvényt tartalmaz, ami a konvertálást egy 6 elemű bufferben tárolja. Ezzel kivédhető az az eset, amikor egy tároló buffer esetén az LCD kiirató függvényben egymás után akarunk több lebegőpontos számot konvertálni, és az egyes konvertálások felülírják az előzőeket. Maximum hat lebegőpontos szám konvertálható egyszerre, és a maximális hosszuk 16 karakter lehet (ami több, mint elég egy 16 karakter széles kijelzőn).

```
char* DSPC_FTS(DSPC_CONV_TYPE f, unsigned char precision) 5.4.2.1.1
```

5.4.2.1.1. char* DSPC_FTS(DSPC_CONV_TYPE f, unsigned char precision)

Függvény: char* DSPC_FTS(DSPC_CONV_TYPE f, unsigned char precision)

Előfeltétel: Nincs

Bemenet: Konvertálandó lebegőpontos szám és a konvertálás felbontása

Kimenet: Karakterbufferre mutató pointer, ami tartalmazza a konvertált számot

Mellékhatás: Nincs

A függvény a paraméternek megadott lebegőpontos számot konvertálja át az adott pontossággal egy karakterbufferbe, aminek a címét visszaadja.

A maximális konvertálási hossz 16 karakter lehet, és egyszerre maximum 6 lebegőpontos szám konvertálható egy függvényhívás alatt (LCD kiirató függvény paramétereként).

5.12. példakód. Lebegőpontos szám kiírása LCD kijelzőre

```
1 // Lebegőpontos szám kiiratása 3 tizedesjegy pontossággal
2 float temp_float = 0.123456f;
3 DSPController_lcd_top("Float: %s",DSPC_FTS(temp_float,3));
4 // Az eredmény: [Float: 0.123 ]
```

6. fejezet

Példa projektek

A DSPController programcsomag részét képezi két minta alkalmazás. Ezekben keresztül bemutatásra kerül az API és a környezetének a használata. A két minta alkalmazás egyforma alapokra épült, csupán az összetettségükben van különbség.

Bare Minimum App	A legalapvetőbb működést prezentáló alkalmazás. Hang-erőszabályzást valósít meg.	6.4.1
Demo App	Komplex, menürendszer vezérelt alkalmazás, jelfeldolgozási és interfész kezelési példakódval.	6.4.2

Mielőtt részleteznénk a két minta alkalmazás pontos működését, ismerkedjünk meg a projekteket felépítő fájlok szerepével!

6.1. Projektek felépítése

A következő szakaszban sorra vesszük a tipikus DSPControllerrel felszerelt alkalmazások fájlrendszerét, a felépítését és az egyes fájlok főbb feladatait.

Projekt fájlrendszer felépítése			
Header files	Core	ad1835.h	6.1.1.1
		tt.h	6.1.1.2
	DSPController	DSPController.h	6.1.1.3
	User	GLUE.h	6.1.1.4
Source files	Core	IRQprocess.c	6.1.1.5
		SPORTisr.c	6.1.1.6
		init1835viaSPI.c	6.1.1.7
		initDAI.c	6.1.1.8
		initPLL.c	6.1.1.9
		initSPORT.c	6.1.1.10
		main.c	6.1.1.11
	DSPController	DSPController.c	6.1.1.12
	User	DSP.c	6.1.1.13
		INTERFACE.c	6.1.1.14

6.1.1. Project fájlrendszer

6.1.1.1. ad1835.h

ad1835.h	Core	Header file	<i>AD1835 driver</i>
----------	------	-------------	----------------------

A fájl definiálja az AD1835 audio codec-hez szükséges összes vezérlési szimbólumot, amelyeket a felkonfigurálás során fog használni az `init1835viaSPI.c` fájl.

Módosítása nem javasolt.

6.1.1.2. tt.h

tt.h	Core	Header file	<i>SHARC központi header fájl</i>
------	------	-------------	-----------------------------------

Mined SHARC processzorra íródott példakód, amit az Analog Devices adott ki, tartalmaz egy központi header fájlt, amit minden fájlhoz beincludeolnak. Ez a fájl tartalmazza az összes globálisan definiált szimbólumot, globális változók definícióját és a közös függvények prototípusát.

Módosítása nem javasolt.

6.1.1.3. DSPController.h

DSPController.h	DSPC	Header file	<i>DSPController driver header fájlja</i>
-----------------	------	-------------	---

A DSPController driver header fájlja, benne definiálva az összes beállítható paraméter, esemény elnevezések, állapotgépek állapotai, valamint minden függvény prototípusa, mind a core egységhez, mind a kiegészítő függvényekhez.

Módosítása nem javasolt.

6.1.1.4. GLUE.h

GLUE.h	User	Header file	<i>INTERFACE.c és DSP.c fájlok összekötése</i>
--------	------	-------------	--

Header fájl, ami összeköti a `DSP.c` és az `INTERFACE.c` fájlokat globális változókkal. Ez az egyik fájl a három közül, amit a felhasználói szintű programozáskor módosítani kell.

6.1.1.5. IRQprocess.c

IRQprocess.c	Core	Source file	<i>Másodlagos megszakítások kezelése</i>
--------------	------	-------------	--

Kiegészítő megszakítások lekezelését tartalmazó fájl. Valós szerepe nincs az alkalmazásban, mert a feladatát kiváltotta a DSPController rendszer. Eredetileg a SHARC kártyán lévő nyomógombokkal valósított meg hn3 hangerőszabájzást.

Módosítása nem javasolt.

6.1.1.6. SPORTisr.c

SPORTisr.c	Core	Source file	<i>SPORT vonalakon érkező megszakítások</i>
------------	------	-------------	---

Az audio codec-ről érkező megszakítások lekezelési helye. Itt történik meg a fixpontos minták lebegőpontossá konvertálása, a jelfeldolgozó függvény hívása, majd a minták visszakonvertálása és kimeneti regiszterekbe töltése, ahonnan a következő megszakításkor kiküldésre kerülnek. Ebben a fájlból hívódik meg a DSPController driver frissítő függvénye is, ami a driver állapotgépét hajtja. Módosítása nem javasolt.

6.1.1.7. init1835viaSPI.c

init1835viaSPI.c	Core	Source file	<i>SPI driver a AD1835 AD konverterhez</i>
------------------	------	-------------	--

Az AD1835 audio codec felkonfigurálása SPI kommunikáción keresztül. A felkonfigurálás után a driver elengedi az SPI hardvert, ezért nem akad össze a DSPController kommunikációjával. Módosítása nem javasolt.

6.1.1.8. initDAI.c

initDAI.c	Core	Source file	<i>DAI kapcsolatok konfigurációja</i>
-----------	------	-------------	---------------------------------------

SHARC processzor szabadon konfigurálható IO lábait köti be a megfelő helyekre a processzoron belül és kívül is. Ezt a módszert a DSPController driver is használja. Ennek az IO funkcionak a segítségével valósítja meg az SPI buszon az interfész eszköz kiválasztását.

Módosítása nem javasolt.

6.1.1.9. initPLL.c

initPLL.c	Core	Source file	<i>PLL egység felkonfigurálása</i>
-----------	------	-------------	------------------------------------

Processzor belső órajelét állítja be 331.776 MHz frekvenciára.
Módosítása nem javasolt.

6.1.1.10. initSPORT.c

initSPORT.c	Core	Source file	<i>SPORT rendszer felkonfigurálása</i>
-------------	------	-------------	--

A SHARC DSP belső adatátviteli buszait konfigurálja fel a minták továbbítására az audio codec és a DSP mag között.
Módosítása nem javasolt.

6.1.1.11. main.c

main.c	Core	Source file	<i>Fő inizializálási és belépési pont</i>
--------	------	-------------	---

A projekt belépési pontja. Itt történik meg a rendszer felkonfigurálása, majd a végtelen ciklusban itt hívódik meg újra és újra alacsony prioritási szinten az interfész update függvénye.
Módosítása nem javasolt.

6.1.1.12. DSPController.c

DSPController.c	DSPC	Source file	<i>DSPController driver implementációs fájl</i>
-----------------	------	-------------	---

DSPController driver implementációt tartalmazó fájlja. Ez tartalmazza az összes működéshez szükséges funkcionálitás implementációját, valamit a hozzájuk tartozó változókat is.
Módosítása nem javasolt.

6.1.1.13. DSP.c

DSP.c	User	Source file	<i>Jelfeldolgozási algoritmus helye</i>
-------	------	-------------	---

Ez a fájl tartalmazza a felhasználó által implementálálandó jelfeldolgozási algoritmust futtató függvényt, amit a rendszer magas priorítású megszakításból hív meg a beérkező minták ütemében.

6.1.1.14. INTERFACE.c

INTERFACE.c	User	Source file	<i>Interfész viselkedéséért felelős fájl</i>
-------------	------	-------------	--

Hasonlóan a DSP.c fájlhoz, ennek a fájlnak is a felhasználó kell kitöltsse a vázát, hogy a kívánt interfész viselkedést elérje. Tipikusan állapotgép segítségével oldjuk meg a feladatot ebben a fájlból.

6.2. Módosítandó és nem módosítandó fájlok

A fájlok leírásából látható volt, hogy a felhasználónak a legtöbb esetben csupán három fájlból kell módosításokat végeznie ahhoz, hogy a kívánt funkcionálitást létrehozza. Ez a három fájl a

- INTERFACE.c
- GLUE.h
- DSP.c

A többi fájl módosítás nem tiltott, de csak akkor javasolt, ha kellőképpen átlátja a felhasználó a rendszer működését. A rendszer mintákat kezelő mechanizmusa érzékeny a módosításokra, könnyű elrontani, ha a kellő ismeretek nélkül nyúlunk hozzá.

6.3. Tipikus programozás menete

A programozáshoz először is tisztázni kell, hogy melyik feladat melyik fájlba kerüljön. Külön kell választani a jelfeldolgozási és az interfész feladatokat.

Tipikusan interfész feladatnak minősül minden olyan kód részlet, amiben a DSPController API-t használjuk. Az API függvényeket tilos magas priorítású szakaszból hívni (azaz a DSP.c fájlból). Jelfeldolgozási feladatnak minősül, és ezért a DSP.c fájlból a helye minden mintákkal operáló kódnak. Fontos megjegyezni, hogy az INTERFACE.c fájlból az update függvény *nincs* szinkronban a jelfeldolgozó rendszerrel, nála alacsonyabb prioritási szinten fut.

Ha a jelfeldolgozási és az interfész oldal között információt kell átvinni, akkor az átvitelhez közös változók kellenek. Ezeket a változókat *GLUE* változóknak nevezzük, és a GLUE.h fájlból

definiáljuk őket extern változóként, majd az `INTERFACE.c` fájlban deklaráljuk őket. Ezek után használatauk minden oldalon lehetséges, információ átvitelére alkalmasak.

Információ lehet például az interfész oldalról a jelfeldolgozó oldal felé a hangerő aktuális értéke, amit az interfész generál a DSPController API segítségével. A másik irányba is elképzelhető az információátvitel. Ha olyan alkalmazásunk van, ahol valamilyen mennyiséget mérni kell majd kijelezni, akkor a mérést a jelfeldolgozási oldalon végezzük, a mért eredményt átadjuk az interfész oldalnak egy *GLUE* változón keresztül, majd az interfész oldalon megvalósítjuk a kijelzését.

A *GLUE* változók definiálása után az interfész és a jelfeldolgozási oldal funkcionalitásainak az implementálása következik. Tipikus megoldás az állapotgépek használata. Ekkor egy állapotváltozó szerint választjuk szét az egyes feladatrészeket egymástól. Az állapotok között jól definiált átmenetek vannak, amiket általában felhasználói beavatkozásra, a DSPController API által generált események indítanak be.

Állapotgépeket alkalmazhatunk minden oldalon. Az interfész oldali állapotgép általában az egyes LCD képernyőket tekinti állapotának, a jelfeldolgozási oldalon pedig a különböző típusú fel dolgozó algoritmusokat. Például egy multieffekt esetében, ami késleltetésen alapuló zenei effekteket valósít meg, egy egy állapot lehet a Delay, Chorus és Flanger effekt, amelyek között az interfész modul kapcsolat *GLUE* változók segítségével.

6.4. Példakódok működése

Az előző részekben megismertük a projektek felépítését, a benne szereplő fájlokat, valamint az alkalmazás megtervezésének az általános logikáját. A továbbiakban megnézzük, hogy mindez hogyan is valósul meg a gyakorlatban a két DSPController csomaghoz tartozó minta alkalmazásban keresztül.

Minden alkalmazás elérhető a GitHub rendszeréből a következő linkekben:

- <https://github.com/tiborsimon/DSPController-BareMinimumApp>
- <https://github.com/tiborsimon/DSPController-DemoApp>

A projektek szabadon letölthetők és módosíthatóak akár offline, akár online is *Fork Request* segítségével. A forráskódban a felhasználó által érdekelt három fájl bőséges kommentezéssel segíti a fejlesztő munkáját. A minta alkalmazások közül a *Bare Minimum App* lehet a kiindulási projektje a saját fejlesztéseknek, míg a jóval összetettebb *Demo App* szemléltető példaként jobban megállja a helyét.

6.4.1. Bare Minimum App

A két minta alkalmazás közül az egyszerűbb. Feladata bemutatni a minimális struktúrájú alkalmazást, ami kihasználja a DSPController API lehetőségeit. Az implementált funkció egy hangerőszabályzó alkalmazás, ami a bemenetére érkező sztereó jelet egy, a felhasználó által beállított értékkel beszorozza, majd minden kimenetre kiküldi azokat. Ehhez az interfészen egy enkódert használ, ami 0 és 60 közötti intervallumban állítja a skálázó tényezőt.

Vegyük sorba az egyes fájlok feladatát és forrását! A forráskódokból a jobb átláthatóság kedvéért nincsenek feltüntetve a kommentezések, amelyek az elérhető forrásokban természetesen szerepelnek.

6.4.1.1. GLUE.h

A feladat elemzésekor kiderült, hogy egy *GLUE* változó el tudja látni az aktuális hangerő értékének az átvitelét az interfész és a jelfeldolgozó fájlok között. A listázott forráskódba ennek a változónak a definíciója látható.

6.1. példakód. GLUE.h megvalósítása

```
1 #ifndef _GLUE_H_
2 #define _GLUE_H_
3
4 #include "tt.h"
5 #include "DSPController.h"
6
7 extern float GLUE_volume;
8
9 #endif
```

6.4.1.2. DSP.c

A jelfeldolgozási feladat nagyon egyszerű ebben az esetben. A bejövő jobb és bal csatornát kell skálázni a *GLUE* változóval átvitt tényezővel, majd az így kapott hangmintákat a kimentekre kell rávezetni.

6.2. példakód. DSP.c megvalósítása

```
1 #include "GLUE.h"
2
3 void process(void) {
4
5     // Változók előkészítése
6     float left = inLeft;
7     float right = inRight;
8
9     // DSP algoritmus
10    left *= GLUE_volume;
11    right *= GLUE_volume;
12
13    // Kimeneti változók feltöltése
14    out1Left = left;
15    out1Right = right;
16    out2Left = left;
17    out2Right = right;
18    out3Left = left;
19    out3Right = right;
20    out4Left = left;
21    out4Right = right;
22}
```

6.4.1.3. INTERFACE.c

A három felhasználói fájl közül a interfész megvalósító fájl felépítése a legbonyolultabb. Ebben a fájlban kap helyet a DSPController API inicializálása, az interfész és a *GLUE változók* deklarálása, valamint az interfész megvalósító állapotgép, ami jelen esetben egy állapotot tartalmaz.

A fájl elején segítő szimbólumokat definiálunk, majd következnek a lokális és *GLUE változók*. Ezt követi az API inicializálása, ahol 48 kHz mintavételi frekvenciát és enkóder sebesség figyelést állítunk be. A további forráskód a harmadik enkóderről beolvastott érték alapján növeli vagy csökkeneti a hangerő értékét, majd a kapott értéket kiírja az LCD kijelzőre, és lebegőpontos számmá konvertálja, majd átadja a *GLUE változónak*.

6.3. példakód. INTERFACE.c megvalósítása

```

1 #include "GLUE.h"
2
3 // Szimbólumok definiálása
4 #define DEFAULT_GLUE_VOLUME 0.5f
5 #define MAX_VOLUME 60
6
7 // Lokális változók
8 int volume = 30;
9
10 // GLUE véltozók deklarációja
11 float GLUE_volume = DEFAULT_GLUE_VOLUME;
12
13
14 // Inicializálás
15 void initInterface(void) {
16     DSPController_init( FS_48KHZ | ENCODER_VELOCITY_ON );
17 }
18
19 // Interfész periodikus frissítése
20 void updateInterface(void) {
21
22     int temp_volume = DSPController_get_encoder(3);
23
24     if(temp_volume != 0) {
25         volume += temp_volume;
26
27         if (volume < 0) volume = 0;
28         if (volume > MAX_VOLUME) volume = MAX_VOLUME;
29
30         GLUE_volume = (float)(volume)/(float)MAX_VOLUME;
31     }
32
33     DSPController_lcd_top("Bare minimum app");
34     DSPController_lcd_bottom("Volume [%02d]",volume);
35 }
```

6.4.2. Demo App

Mivel a Demo App minta alkalmazás egy viszonylag összetett szerkezetű program, ezért működésének a teljesen részletekbemenő tárgyalásától eltekintünk és helyette egy átfogó képed alkotunk az egyes fájlok feladatáról. A program összetettségét szemlélteti, hogy a felhasználói forráskód több, mint 800 soros, ezért a teljes kód közlésétől is eltekintünk, csupán kód részleteken mutatjuk be a fontosabb működési egységeket. A teljes forráskód elérhető a GitHub-on.

Az alkalmazás képességei

- Bemutatja a DSPController API minden eszközének a használatát
- Szemlélteti a *GLUE változók* logikus felhasználását
- Bemutatja az állapotgépek működtetését
- Tényleges jelfeldolgozási feladatot valósít meg (rezonáns aluláteresztő szűrő)

6.4.2.1. Működés

Az alkalmazás indulásakor egy kezdőképernyő fogad, ami megfelel az interfész kezelő állapotgép első állapotának. Ebből az állapotból bármely gomb lenyomása átviszi a második állapotba, ahol választani lehet az audio effekt és az interfész bemutató program között.

A navigáció innen kezdve a navigációs gombokkal (*A1-A5*) történik. Az egyes képernyőkön belül a jobbra balra mozgást a két szélső gombbal (*A2, A4*) tehetjük meg. Nyugtázo gomb a középső (*A3*), és vissza gomb a felső (*A1*).

Válasszuk ki az interfész bemutatóját (Interface demo), és nyomjuk meg a kiválasztó gombot. A képernyő vált, és az interfész bemutató program lép működésbe (ami program szempontjából megfelel egy állapotgép váltásnak, az interfész bemutató állapot kapja meg a vezérlést). Ebben a bemutatóban az enkóderek tekerésével állíthatjuk a LED oszlopokat, lenyomásukkal a képernyőn kiválasztódik az adott enkóder értéke. A numerikus billentyűzettel bevihetünk számokat a felső LCD sorra. Ehhez az Assembler modul nyújtja a szoftveres támogatást. A lebegőpontos számok kiírásánál a DIP kapcsolókkal adható meg a kiírás pontossága. Legalább egy tizedesjegy pontos a kijelzés, majd ahány kapcsoló van lekapcsolva alulról felfelé folyamatosan, annyival nő a kijelzett tizedesjegyek száma.

A programból kilépni a felfelé gombbal (*A1*) lehet.

A program semmi értelmes feladatot nem lát el, egyedüli célja, hogy a fejlesztőnek bemutassa a DSPController API képességeit, és egy példát adjon az alapszintű programozására.

Ha visszalépünk egyet, ismét a választó képernyőre kerülünk, válasszuk az Audio effect-et. Ekkor ismét egy választó képernyőre kerülünk, ahol kiválaszthatjuk az effekt típusát az egyes enkóderrel, és beállíthatjuk a kimeneti multiplexer jelvezetését a hármas enkóderrel. Az Effect beállítása alap-értelmezetten *Mute*, azaz nem ad a kimenetre semmilyen mintát a jelfeldolgozó program. Az egyes enkóder tekerésével választható még *Osc*, mint oszcillátor és *LPF*, mint aluláteresztő szűrő. Ha a *Mute* értékről eltekerünk, látható, hogy megjelenik az Effect körül egy keretezés. Ez jelzi, hogy a kiválasztó gombbal (*A3*) beléphetünk további beállításokért. A Effect értékeinek a változtatása közben feléled a SHARC kártya kimenete, és a kiválasztott effect típusának megfelelő kimeneti jelet produkál:

- *Osc*: 440 Hz-es színeszös mono jelet generál.
- *LPF*: A kártya bemenetére adott jelet szűri meg az aluláteresztő szűrő alapbeállításával.

A kimeneti multiplexer jelvezetése szabadon állítható minden beállítás mellett. A SHARC kártyának 4 kimenete van, ezen kimenetek kombinációi közül lehet választani a harmadik enkóder segítségével. A kijelzett négy bináris érték a négy kimeneti vonalnak felel meg. 0 esetén a kimenet inaktív, 1 esetén a kimenetre rá van vezetve az effect modul jele.

Válasszuk ki az *Osc* beállítást az egyes enkóderrel, majd lépjünk bele a beállításaiiba a kiválasztó gombbal. A megjelenő képernyőn három beállítási lehetőség van a három enkóderhez rendelve:

- *Type*: Generált jel típusa: [színeszös|négyszög]. (triviális szintetizálás, nem lényeges a jó minőségű jel generálása)
- *Freq*: Generált jel frekvenjiája.
- *Amp*: Generált jel amplitúdója.

Visszalépve az előző menüpontba válasszuk az *LPF* effectet és lépjünk be a beállításaihoz. Itt egyből feltűnhet a képernyő jobb oldalán a nyilacska, aminek jelzi, hogy több beállítható paraméter is van, elérésükhez lapozni kell a menüben a navigációs gombok jobbra és balra nyilaival (*A2,A4*). A lehetséges paraméterek:

- *Fc*: Szűrő vágási frekvenciája.
- *Q*: Szűrő kiemelése a vágási frekvenciánál.
- *Mod*: Vágási frekvenciát moduláló jel típusa: [színeszös|négyszög].
- *Span*: Moduláció mélysége. *Fc*-től mennyire téritse ki a frekvenciát a modulátor jobbra és balra.
- *MFreq*: Moduláció frekvenciája.

Ha játszunk kicsit a beállításokkal, akkor észrevehetjük az *Fc* és a *Span* paraméter kapcsolatát. A moduláció mélysége nem lehet nagyobb, mint a vágási frekvencia, különben negatív frekvenciába modulálná az effektet, viszont a ugyan így a vágási frekvencia sem lehet kisebb, mint a modulációs mélység. Ezeket a feltételeket a program betartja. A hasonló megkötésekért érdemes tervezés alatt felismerni, mert ha fejlesztés közben derül ki, hogy a szűrő képes összeomlani szerencsétlen paraméterkombinációjánál, akkor kellemetlen, percekig tartó fülcsengés lehet az eredménye.

6.4.2.2. Szoftveres fogások

Az eddigiekben megismertük a Demo App alkalmazás felhasználói szintű működését. A mostani alfejezetben kiemelünk néhány programozási fogást, ami hasznos lehet a későbbi fejlesztések során. Az egyes részekben közölt forráskód csak felületes tudást nyújt, és nem váltja ki a teljes forráskód tanulmányozása során kialakuló képet.

6.4.2.2.1. GLUE változók használata

A Demo App-ban bemutatásra kerül a *GLUE változók* célszerű alkalmazása. A módszer hasonló a már eddig használtakhoz, a különbség a kezdeti érték megadásában van. A következő kód részletben kiemeljük a lényeget.

6.4. példakód. INTERFACE.c megvalósítása

```
1 // A pédkód az alkalmazás aluláteresztő szűrőjének a vágási
2 // frekvenciáján mutatja be az eljárást
3
4 // GLUE.h fájlban
5 ...
6 // Alapértelmezett érték definiálása
7 #define DEFAULT_GLUE_2_fc 1000.0f
8 ...
9
10 // GLUE változó definiálása
11 extern float GLUE_1_freq;
12 ...
13
14
15 // INTERFACE.c fájlban
16 ...
17 // GLUE változó deklarálása és alapértelmezett értékre beállítása
18 float GLUE_2_fc = DEFAULT_GLUE_2_fc;
19 ...
```

Látszólag túl lett bonyolítva a változó deklarálása, de egy jelentős előnye van ennek a módszernek: ha később változtatni szeretnénk a kezdeti értéket, azt elszeparálva az implementációs fájltól tehetjük meg a változó definiálásánál. Ennek az előnye nagy méretű projektek nél mutatkozik.

6.4.2.2. Menu pointer

A forráskódban látható `menu_pointer` változó egy állapotváltozó az egyes állapotokon belül. Használata akkor javasolt, ha a képernyőn a felhasználó ki tud választani egy vagy több elemet, és lépegetni is tud köztük. A mutató (ami a szó szoros értelmében nem mutató) ekkor az aktuális elemre mutat, és szerepe a képernyő renderelésében és a továbblépés irányának a meghatározásában van.

Az alkalmazásban egy globális `menu_pointer` szerepel, de szofisztikáltabb menürendserekben ajánlott a minden állapothoz külön `menu_pointer` használata, mert így a menü szintjei között lépkedve eltárolható, hogy az adott szinten melyik menüelem volt fókuszban. Ez megkönnyíti a navigációt, mert a felhasználó visszalépésnél oda kerül vissza, ahonnan kiindult, így nem keveredik el.

6.4.2.2.3. Állapotváltások közötti eseménytörlés

Mint az API leírásában is olvashattuk (5.3.4), érdemes állapotok közti váltáskor törölni a nem feldolgozott eseményeket a `DSPController_flush()` függvényvel, hogy ne változtassák meg az első pillanatban a következő állapot beállítását, ha abban felhasználjuk az adott eseményeket.

Egy egyszerű példa erre egy olyan alkalmazás, ami a Demo App-hoz hasonlóan egy kezdő képernyővel kezd, de csak egy specifikus gomb lenyomására lép tovább a következő állapotba, ahol az enkóderekkel állítható be például a feldolgozott csatornák hangereje. Ha a felhasználó az első képernyőn eltekeri az enkódereket (márpedig semmi se akadályozza meg ebben), és az kezdőkép-

ernyő állapota nem kérdezi le az enkóderek változását, akkor a driver akkumulálja a tekeréseket, majd mikor a program állapotot vált, az első enkóder lekérdezésnél akár egy nagy értéket is kiolvashat, amit felhasználva akár kellemetlenül hangos vagy halk kezdőhangero is kialakulhat, holott a felhasználó nem is számított rá. Ha töröljük az eseményeket, ilyen hiba nem fordulhat elő.

6.4.2.2.4. Állapotok elnevezése

Egy állapotgéppben az egyes állapotokhoz számok vannak rendelve, ez alapján történik meg a vezérlés megfelelő kiosztása az egyes ágak között. Ez azonban nem jelenti azt, hogy a programban is számokkal kell jelölni az állapotokat. Jó programozási gyakorlat, hogy minden egyes állapotra definiálunk egy beszédes szimbólumot, amivel a program, de legfőbbképpen a programozó kiigazodik a forráskódban.

Általános gyakorlat, hogy a definiált szimbólumokat csupa nagybetűkkel írjuk, a nevük pedig balról jobbra az általánostól az egyre specifikáltabb elnevezéshez tendál.

Ezt a logika használatos a teljes DSPController API és a minta alkalmazások esetében is.

Példaként vegyük a DSPController API esemény elnevezési konvencióját: DSPC_EVENT_E2_LONG.

Balról jobbra haladva, először a DSPC jelöli, hogy a DSPController API egy szimbólumával van dolgunk. Utána az EVENT jelzi, hogy egy esemény definiálása lesz, majd az E2 specifikálja, hogy melyik gombhoz tartik az eseményt, végül a LONG definiálja teljesen, hogy pontosan mi is történt. Ez a felülről lefelé, nagyobb halmaztól a kisebb felé logika jól áttekinthetővé teszi a forráskódban alkalmazott szimbólumokat. Használata nem kötelező, de nagyobb fejlesztések esetén javasolt.

6.4.2.2.5. GLUE vagy nem GLUE változó

A program tervezésénél érdemes szétválasztani, hogy mi kerül a *GLUE változók* közé és mi nem. Mivel többségében nem időkritikus az interfész kezelése, így a globális és lokális változók elérési idejének különbsége nem játszik akkora szerepet, viszont ha minden változót *GLUE változónak* definiálunk, akkor elveszik a jól szeparált programozási képünk. Az alkalmazás működni fog, de ha később módosítani szeretnénk, akkor lehet nem lesz egyértelmű, hol is nyúljunk hozzá.

Felmerülhet a kérdés, hogy egy interfészen kijelzett értéket, amit a DSP algoritmus is felhasznál, érdemes-e közös *GLUE változóba* tenni. A válasz erre az, hogy ha csak kijelezzük, a DSP kód által frissített változót, akkor elegendő egy *GLUE változót* használni a feladatra, de ha az interfész módosítani is tudja a változót, akkor mindenkorban szükség van egy lokális segédváltozóra, ami az egész számokkal működő DSPController API számára reprezentálja a paramétert, amit majd a megfelelő algoritmus átkonvertálja a DSP kódhoz használható lebegőpontos paraméterré. Ez alól kivételt képez, ha az Assemblert használjuk (5.4.1), ami lebegőpontos értékekkel tér vissza.

7. fejezet

További fejlesztési lehetőségek

7.1. Ismert hibák

A rendszer nem teljesen hibamentes. A fejlesztés során több furcsak jelenséggel találkoztam a SHARC kártyán való dolgozás során, melyek oka a SHARC DSP viszonylagos bonyolultsága és a pontos kiismerésére való idő szűkösségeinek a kombinációja. A felmerülő problémákból egy maradt megoldatlan, részben azért is, mert a debuggolása szinte lehetetlen, mivel az SPI kommunikációs csatornát kéne hosszabb időn keresztül megfigyelni, miközben figyeljük magát az eszközt is, hogy mit csinál a belső kommunikációs állapotgépe.

A hiba akkor mutatkozik, ha hosszabb ideig nem frissítünk egy LCD sort, míg a másik sort aktívan használjuk. Ekkor valami oknál fogva képesek összekeveredni, a felső és alsó sor üzenetei, és a statikus sor tartalma felülíródik a gyorsan frissített sor egy adatcsomagjával. A hiba véletlenszerűen jelentkezik, nem tudtam semmilyen belső mechanizmushoz társítani. Legtöbbször akkor jelentkezett (de akkor is elég ritkán), ha nagyon gyorsan tekert enkóder értékeit irattam ki az LCD-re. Lassab tekerésnél szinte sosem fordult elő.

Egy lehetséges megoldás lehet az, ha az LCD kezelő modult úgy írjuk át, hogy ha a felhasználó olyan szöveget szeretne a képernyőre írni, ami megegyezik az aktuális megjelenítettel, akkor ahelyett, hogy nem küldi ki az azonos adatsort, ritkább időközökként küldi ki, ami még nem terheli túl az SPI vonalat. Ezzel ha véletlenül felül is írná a gyorsan frissül LCD sor a statikusat, a felülírás csak egy pillanatig jutna érvényre, utána visszaíródná az eredeti sor tartalma.

7.2. Fejlesztési lehetőségek

Ahogy konzulensem, Orosz György, az utolsó közös átnézésen rávilágított, hasznos lenne, ha integrálva lenne a rendszerbe egy blokkolás, ami a SHARC kártya debuggolásakor letiltja az interfész egység eseményfigyelését, és így elkerülhető lenne a rendszer mostani egyik gyengesége, hogy debuggolás esetén, ha a SHARC kártya nem kérdezi le az eseményeket az eszkösről, mivel a programja áll, és a felhasználó véletlenül megnyom valami gombot, esetleg többször is, akkor az így keletkező események felhalmozódnak, és a SHARC kártya újbóli beindulásánál úgymond elárasztják az interfész programot. Ezzel nem várt állapotba is kerülhetünk, a program futása követhetetlen lenne. A problémára a megoldás igen egyszerű, integrálása a rendszerbe egy következő frissítésnél esedékes.

Az interfész eszköz vezérlésébe kell belenyúni, oly módon, hogy amennyiben egy bizonyos időn belül nem érkezett lekérdezése parancs az SPI buszon (azaz a SHARC kártya működése leállt), letiltja az események eseménybufferbe való írását, majd az újbóli lekérdezés hatására, újra engedélyezi azt.

Irodalomjegyzék

- [1] Arduino. Arduino. <http://arduino.cc/>.
- [2] Atmel. Atmega328p. <http://www.atmel.com/Images/doc8161.pdf>.
- [3] Analog Devices. Adum. http://www.analog.com/static/imported-files/data_sheets/ADUM1410_1411_1412.pdf.
- [4] Analog Devices. *ADSP-21364 EZ-KIT Lite Evaluation System Manual*. AD, August 2006. URL: http://www.analog.com/static/imported-files/eval_kit_manuals_legacy/33168315105663ADSP_21364_EZ_KIT_Lite_Manual_Rev._3.0.pdf.
- [5] Analog Devices. *ADSP-2136x SHARC Processor Hardware Reference*. AD, June 2009. URL: http://www.analog.com/static/imported-files/processor_manuals/ADSP-2136x_HRM_Rev2.0.pdf.
- [6] Soundart. Chameleon. <http://www.chameleon.synth.net/>.
- [7] Simon Tibor. *DSPController Gyors Kezdési Útmutató*. BME MIT, January 2014. URL: <https://github.com/tiborsimon/DSPController-Bundle>.