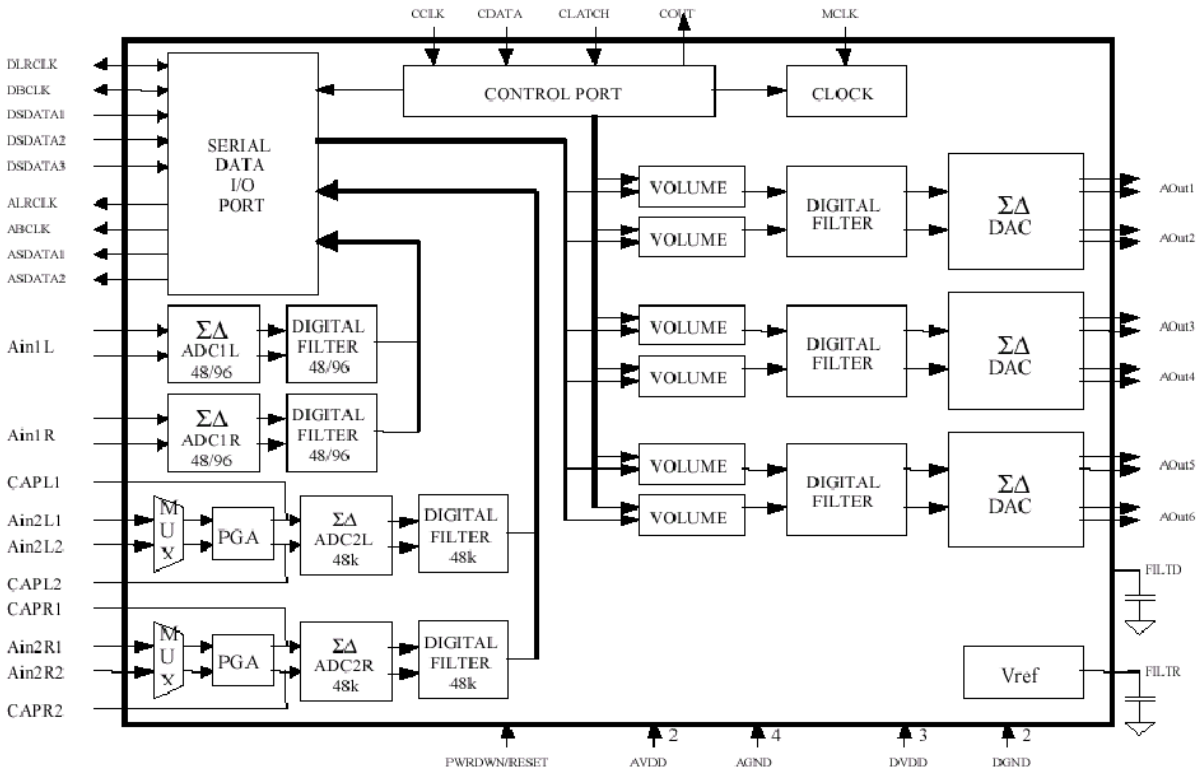
Instrukcja programowania i obsługi codeca AD1836 dla procesora sygnałowego ADSP-21161 SIMD SHARC z wykorzystaniem płytki rozszerzeniowej ADDS-21161-EZLITE

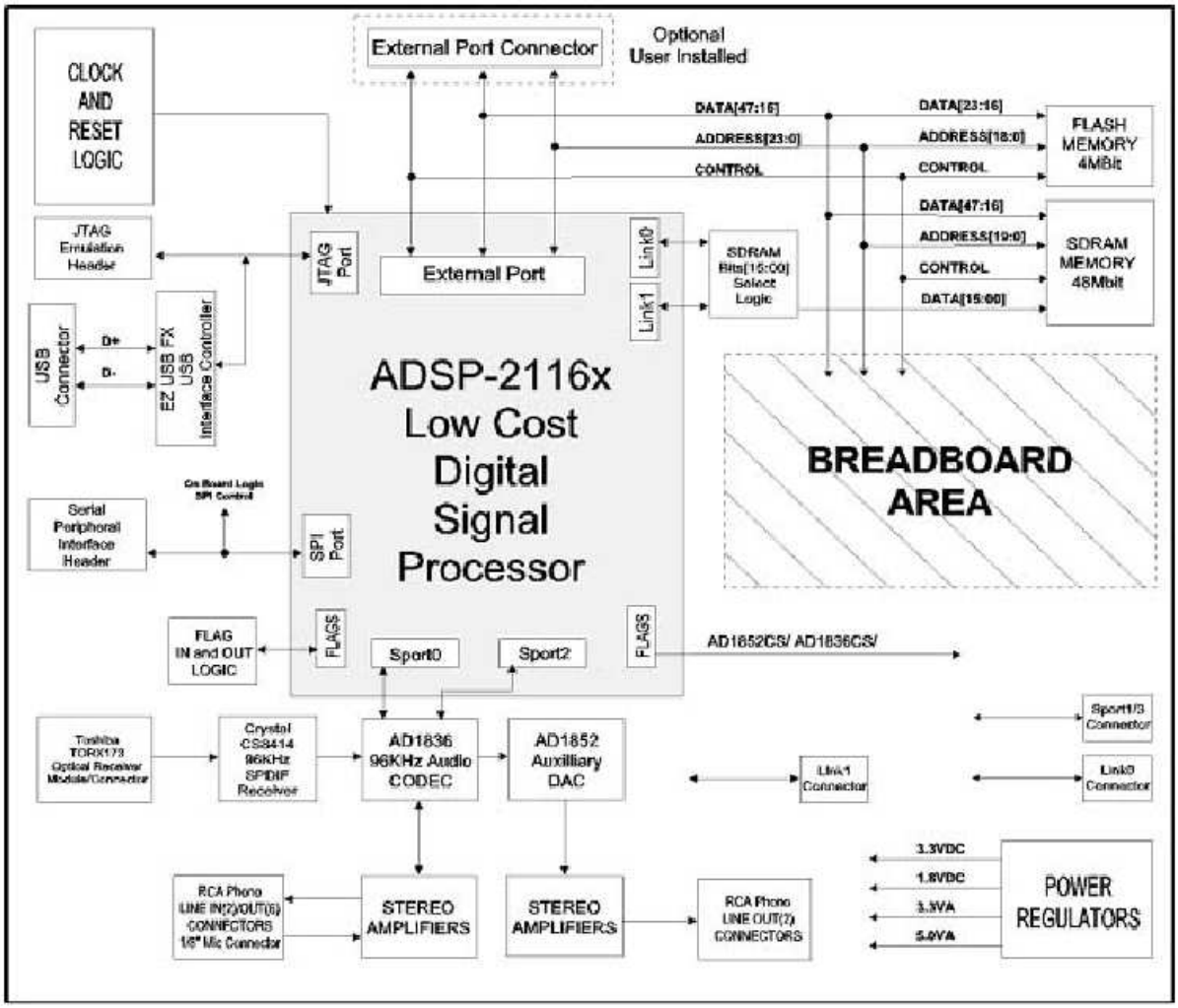
I. Wiadomości ogólne.



Rys 1. Schemat blokowy codeca

Parametry:

- Rozdzielczość: 24 bit
- Częstotliwość próbkowania: 48/96 kHz
- Zakres dynamiki sygnału i SNR: 105 dB
- 3 konwertery C/A stereo, 2 konwertery A/C stereo oparte na modulatorach Δ–Σ
- 400 milionów operacji zmiennoprzecinkowych na sekundę (szczytowo 600)



Rys 2. Schemat blokowy płytki rozszerzeniowej ADDS-21161-EZLITE

Komponenty płytki rozszerzeniowej:

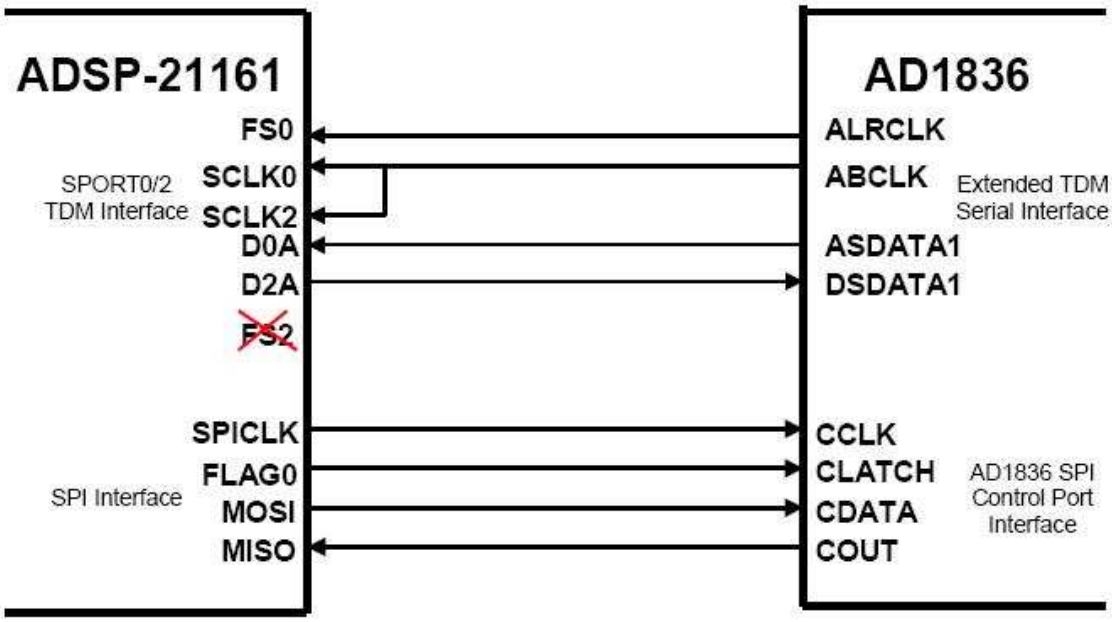
- Procesor sygnałowy ADSP-21161 SIMD SHARC taktowany zegarem 100MHz;
- Pamięć:
  - 1 Mbit pamięci wewnętrznej *on-chip,*
  - 48 Mbit pamięci zewnętrznej SDRAM taktowanej zegarem 100MHz,
  - 4Mbit pamięci typu flash;
- 16-bitowy mikrokontroler Cypress CY7C6403 EZ-USB podłączony do portu JTAG, umożliwiający „nieinwazyjną" emulację interfejsu JTAG poprzez port USB w komputerze PC;
- Codec AD1836;
- Konwerter C/A stereo AD1852;
- Odbiornik typu SP/DIF, Crystal CS8414 (24-bit, 96kHz);
- Złącza:
  - Wejście mikrofonowe stereo,
  - Wejście Line-in RCA,
  - 8 wyjść RCA,
  - Cyfrowe wejście optyczne/RCA audio;
- Złącze emulatora interfejsu JTAG;
- Inne złącza (EP, SPORT (porty szeregowe), Link Ports);
- 6 LEDów, 4 flagowe przyciski Push-Button, 3 przerwaniowe przyciski Push-Button.

II. Szeregowy interfejs między układami AD1836 i ADSP-21161.

AD1836 komunikuje się z procesorem DSP poprzez porty I2S, jednak omówiony zostanie jedynie uproszczony tryb TDM (time-division-multiplexed mode) przesyłania danych.

Za pomocą interfejsu szeregowego układ codeca AD1836 komunikuje się z DSP lub układem ASIC za pomocą trybu TDM, w którym konwertery A/C i C/A przesyłają dane w oddzielnych przedziałach czasowych ramki TDM. Dostęp do rejestru rozkazów/statusu codeca odbywa się przez port kompatybilny z SPI, procesor ADSP-21161 do tego celu wykorzystuje interfejs SPI lub SPORT. **Uwaga! W trybie SPORT TDM układ AD1836 zapewnia próbkowanie sygnału jedynie 48kHz**.

W trybie TDM w skład cyfrowego interfejsu szeregowego wchodzą 4 piny łączące codec z procesorem DSP. Są to: ALRCLK, ABCLK, ASDATA1 i DSDATA1. Wszystkie cyfrowe dane audio są przekazywane tą drogą, natomiast informacje dot. rozkazów i statusu są przekazywane przez interfejs SPI.



Rys.3 Połączenia między układem codeca AD1836 a procesorem DSP ADSP-21161

Tryb TDM układu AD1836 definiuje cyfrowe połączenie szeregowe, będące dwukierunkowym strumieniem cyfrowym kodowanym w systemie PCM. Procesor ADSP-21161 wykorzystuje kilka strumieni audio we/wy, opierając się na schemacie TDM. Architektura codeca dzieli każdą ramkę audio na 8 wychodzących i 8 przychodzących strumieni danych, każdą o rozdzielczości 24bit, zawartych w 32-bitowych przedziałach czasowych.

1. Sygnały zegarowe i synchronizacja ramek.

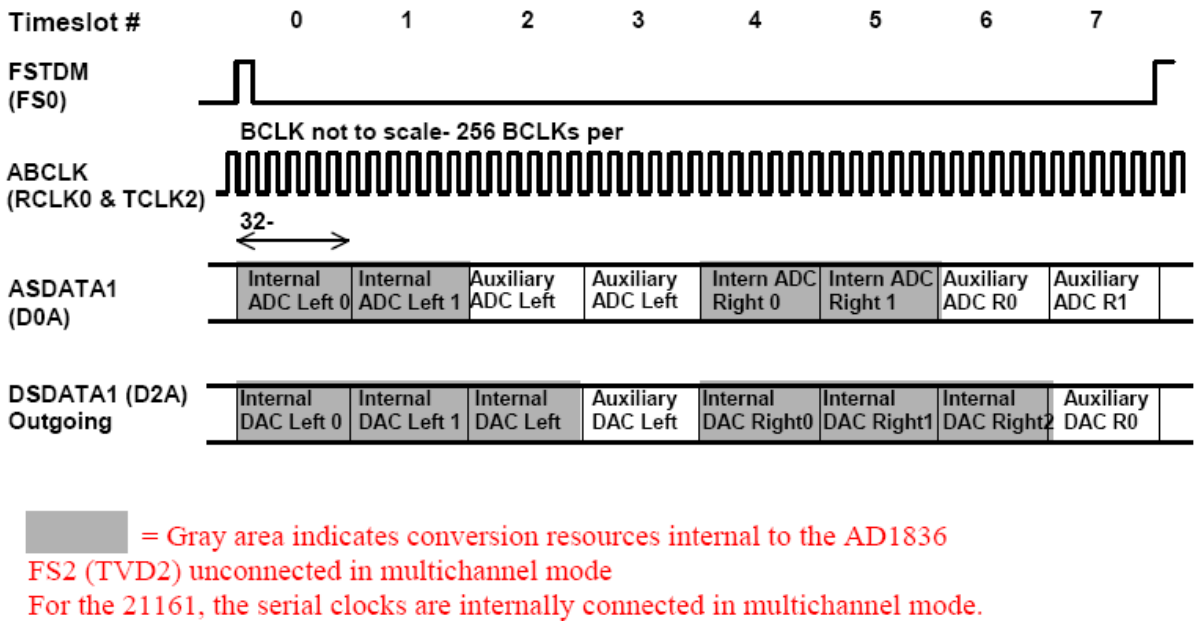
Aby zminimalizować błędy zegara, sygnał taktujący dostarczany jest z zewnątrz z oscylatora kwarcowego o częstotliwości 12.288MHz. Do procesora dostarczany jest buforowany zegar przez szeregowe połączenie ABCLK. Błędy zegara w konwerterach A/C i C/A codeca mają fundamentalny wpływ na jakość sygnału wyjściowego. Ustalony na 12.288MHz zegar ABCLK ma wystarczającą „ziarnistość” dla 8 32-bitowych wejściowych i wyjściowych przedziałów czasowych z częstotliwością próbkowania 48kHz. Szeregowe dane TDM są podawane na każde narastające zbocze zegara ABCLK. Odbiornik danych TDM,

codec dla danych wyjściowych i procesor dla wejściowych, pobiera każdy bit na zbocze opadające. Codec dostarcza dane szeregowe na 12.288MHz, które procesor synchronizuje a następnie tworzy odpowiednie ramki audio.

Początki wszystkich ramek audio (pakietów próbek audio), przenoszonych przez łącze TDM, są synchronizowane z narastającym zboczem zegara FSTDM (ALRCLK). W trybie TDM pin ALRCLK jest zastąpiony pinem FSTDM. Pin ten jest używany do synchronizacji ramek i jest generowany przez codec jako wejście do procesora. Synchronizacja wszystkich transferów danych TDM jest sygnalizowana przez procesor na linii F0. FSTDM, ustalony na 48kHz, jest uzyskiwany poprzez dzielenie zegara ABCLK. Procesor traktuje SCLK0 (ABCLK) i FS0 (FSTDM) jako wejścia. Synchronizacja ramki jest generowana raz na 256 cykli SCLK0, co daje 48kHz sygnał FSTDM, którego okres definiuje ramkę audio. FSTDM (FS0) jest dostarczany z codeca. Aby pogodzić dostarczany z zewnątrz zegar 48kHz z generowanym wewnątrz 12.288MHz SCLK, procesor ustawia w rejestrze SPCTL odpowiednie bity na 1. Częstotliwość podawanej ramki z codeca jest zawsze równoważna częstotliwości próbkowania.

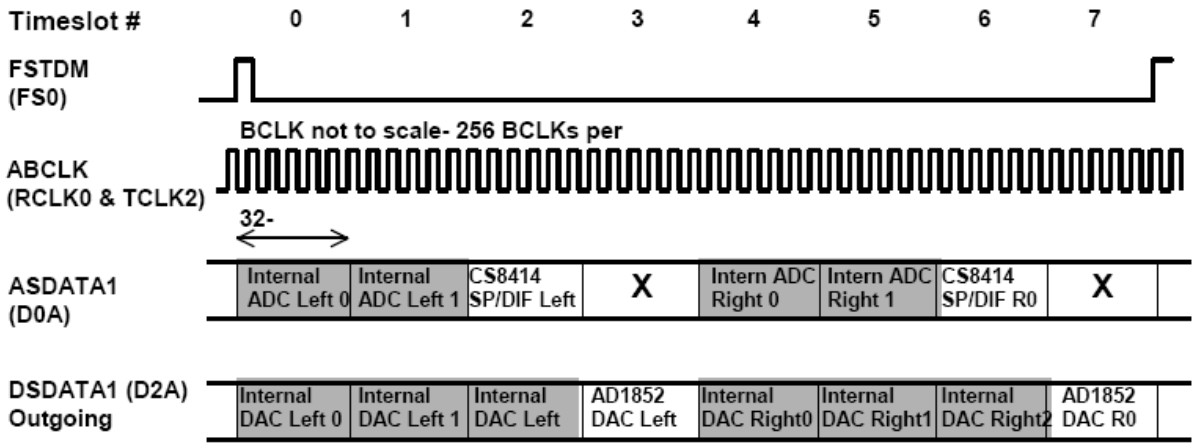
Piny ADSDATA1 i DSDATA1 są szeregowymi wejściami i wyjściami dla danych z codeca. Oba piny przenoszą dane w 8 różnych przedziałach czasowych na jedną ramkę audio. Dane są transmitowane na każde narastające zbocze zegara ABCLK (SCLK0), zaś dane otrzymywane są próbkowane na każde zbocze opadające.

2. AD1836/ADSP21161 EXT-TDM - Protokół cyfrowego interfejsu szeregowego.

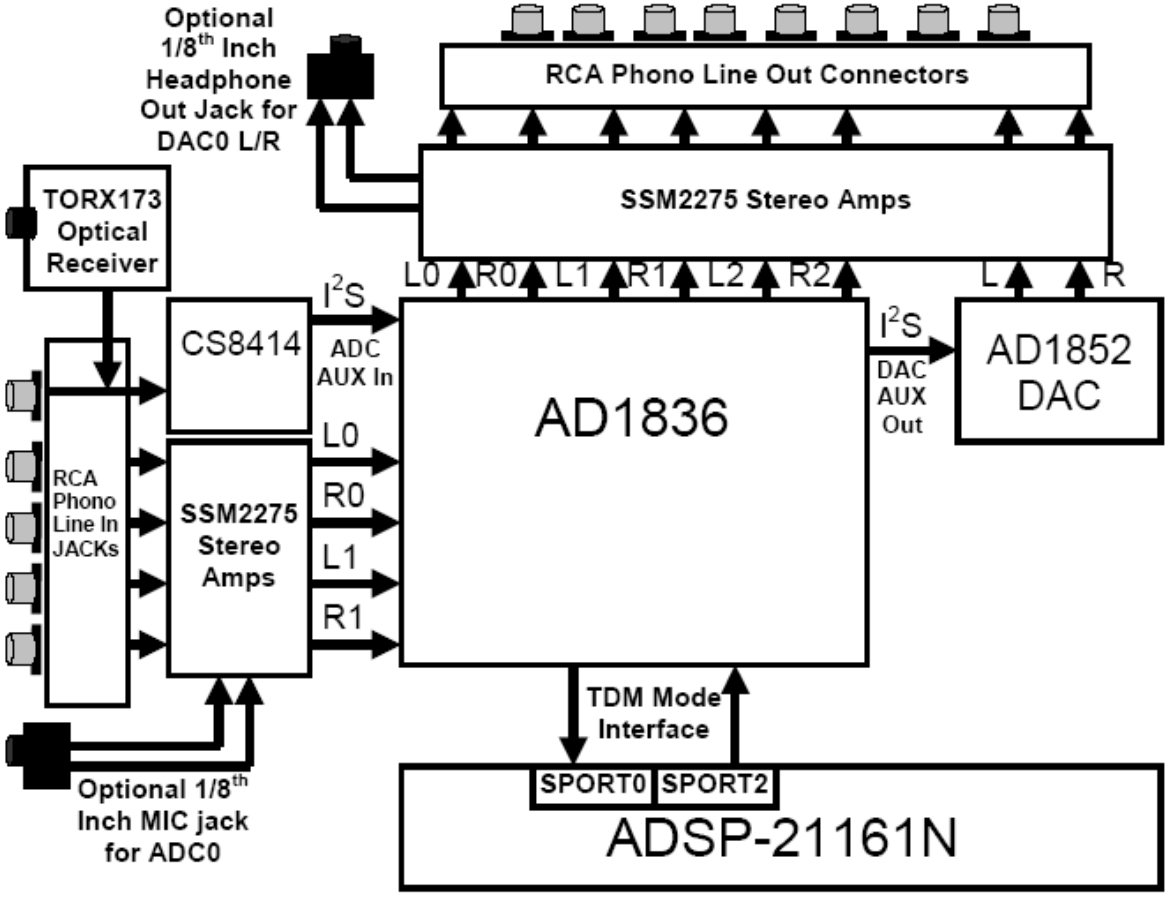


Rys.4 Tryb Rozszerzony TDM – dwukierunkowa ramka audio

Protokół trybu rozszerzonego TDM dla płytki 21161 EZ-KIT Lite korzysta z jednego pomocniczego urządzenia wejściowego – odbiornika CS8414 SP/DIF, oraz z jednego pomocniczego urządzenia wyjściowego – konwertera C/A AD1852.



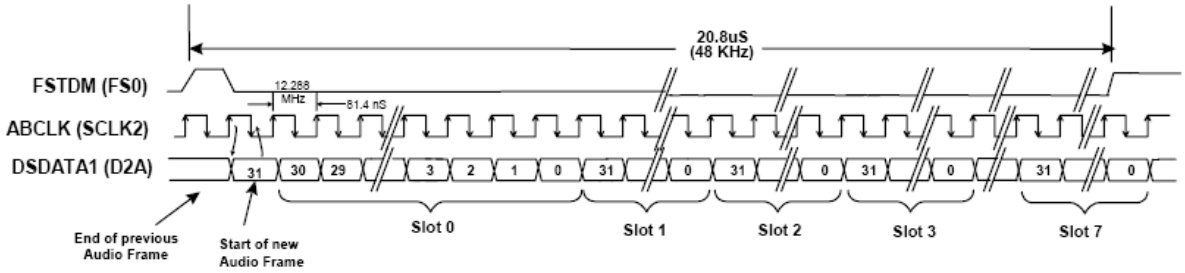
Rys.5 Zmodyfikowany tryb rozszerzony TDM – ramka audio dla płytki 21161 EZ-KIT Lite



Rys.6 ADSP-21161 EZ-KIT Lite Audio Interface – Tryb rozszerzony TDM z dwoma urządzeniami pomocniczymi.

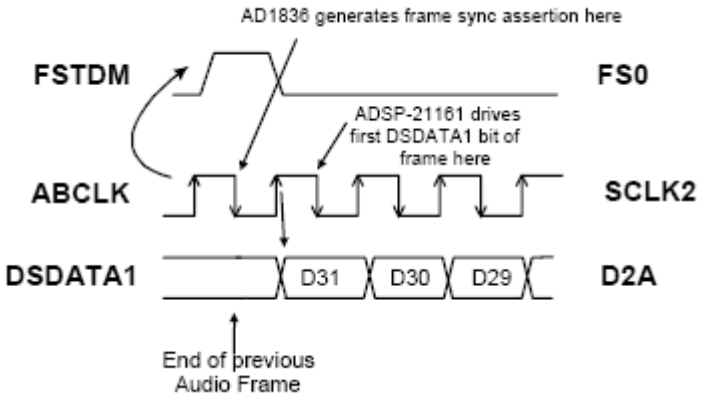
3. Wyjściowa ramka audio układów ADSP21161 / AD1836 (z D2A do DSDATA1)

Wyjściowe strumienie danych (ramki audio) korespondują z multipleksowanymi wiązkami cyfrowych danych wyjściowych zmierzających ku wejściom konwertera C/A codeca. Każda wyjściowa ramka audio może zawierać 8 32-bitowych przedziałów czasowych. Poniższy wykres ilustruje protokół oparty na przedziałach czasowych.



Rys.7 Wyjściowa ramka audio – z DSP do codeca.

Ramka zaczyna się od momentu przejścia FSTDM ze stanu niskiego w stan wysoki. FSTDM jest zsynchronizowany z narastającym zboczem zegara ABCLK. Następujące po tym zbocze opadające oznacza początek nowej ramki audio. Na następne zbocze narastające ABCLK, procesor wysyła pierwszy bit DSDATA1 do przedziału 0 (MSB pierwszy). Każdy kolejny bit jest wystawiony na narastające zbocze ABCLK i pobierany przez codec na zbocze opadające. Ta sekwencja zapewnia, że przesył danych i punkty próbkowania (pobierania)dla strumieni danych wyjściowych i wejściowych są czasowo wyrównane.



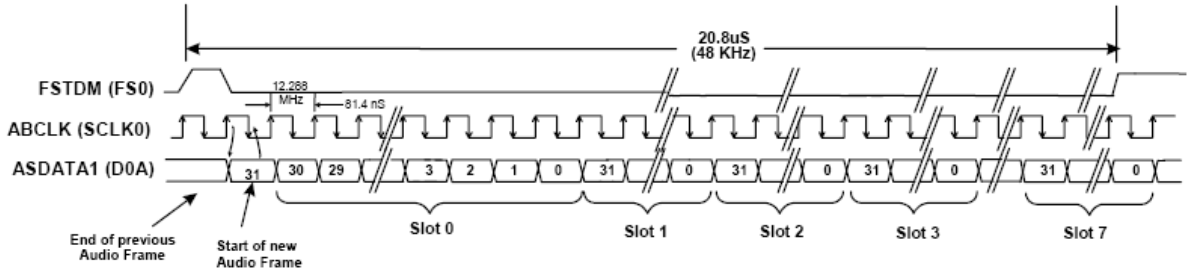
Rys.8 Początek wyjściowej ramki audio.

Złożone strumienie danych z D2A/DSDATA są przesyłane trybem BigEndian (pierwszy MSB), w których wszystkie nieprawidłowe pozycje bitowe w przedziale są wypełniane zerami przez DSP. Oprogramowanie procesora inicjalizuje bufor DMA w sterowniku codeca wypełniając go zerami. 24-bitowe dane audio zawarte w 32-bitowych przedziałach czasowych są „wyrównane do lewej", czyli 24-bitowa informacja przetworzona przez konwerter C/A codeca rezyduje na pozycjach bitowych od 31 do 8. W przypadku gdy w poprawnym przedziale czasowym jest mniej niż 32 bity, procesor wypełnia te niewykorzystane zerami. Gdy na wyjściu procesora jest strumień mono, programista może opcjonalnie zapewnić, że każdy lewy i prawy przedział czasowy będzie wypełniony tymi samymi danymi.

4. Wejściowa ramka audio (z SDATA_IN do DR0)

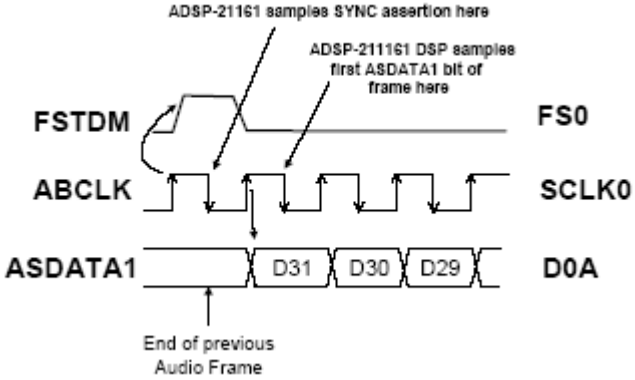
Strumienie wejściowych ramek audio korespondują z multipleksowanymi wiązkami wszystkich cyfrowych danych wejściowych zmierzających do procesora DSP. Podobnie jak dla ramek wyjściowych, ramki wejściowe codeca również składają się z 8, 32-bitowych

przedziałów czasowych (*ang. time slots*). Poniższy wykres ilustruje protokół TDM oparty na przedziałach czasowych.



Rys. 9 Wejściowa ramka audio TDM – z AD1836 do ADSP-21161

Wejściowa ramka audio (próbki danych wysyłanych z codeca do DSP) rozpoczyna się z narastającym zboczem FSTDM (FS0). FSTDM jest zsynchronizowany z narastającym zboczem ABCLK (SCLK0). Jak tylko pojawi się narastające zbocze ABCLK, codec natychmiastowo próbkuje stan FSTDM. Opadające zbocze oznacza moment, w którym obie strony łącza są „świadome" początku nowej ramki audio. Na następne narastające zbocze ABCLK codec wystawia pierwszy bit w przedziale 0 na linii ASDATA1. Każdy nowy bit jest wystawiany na narastające zbocze ABCLK i następnie pobierany przez DSP na zbocze opadające. Sekwencja zapewnia, że przesył w obie strony jest czasowo wyrównany.

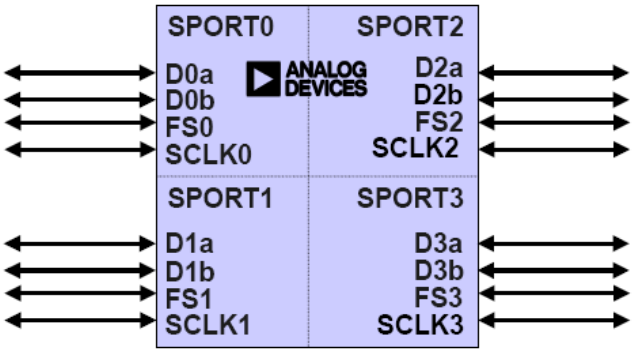


Rys. 10 Początek wejściowej ramki audio

5. Konfiguracja Interfejsu do wielokanałowego portu szeregowego procesora ADSP-21161.

Połączenie między codekiem a procesorem może być zrealizowane zarówno przez parę SPORT0/2 jak i przez SPORT1/3. EZ-KIT Lite korzysta z pary SPORT0/2.

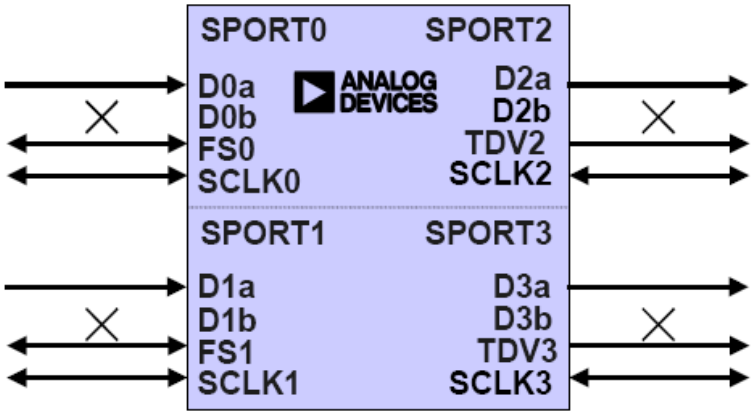
Szeregowe port codeca i DSP przesuwają dane począwszy od MSB, zaś częstotliwość zegara codeca ABCLK (12,288 MHz) jest mniejsza od maksymalnej częstotliwości zegara DSP SCLK (50 MHz). Stąd częstotliwość zegara rdzenia procesora (CCLK) musi być większa niż 12,288 MHz.



Rys. 11Piny SPORT procesora ADSP-21161

Szeregowy port (SPORT) procesora ma dwa piny danych, które mogą być skonfigurowane jako odbiornik lub nadajnik. Piny te są dwukierunkowe i mogą być programowane przez ustawienie bitu DDIR (Data Direction) w SPCTL.

- Kanały A – D0A, D1A, D2A, D3A
- Kanały B – D0B, D1B, D2B, D3B

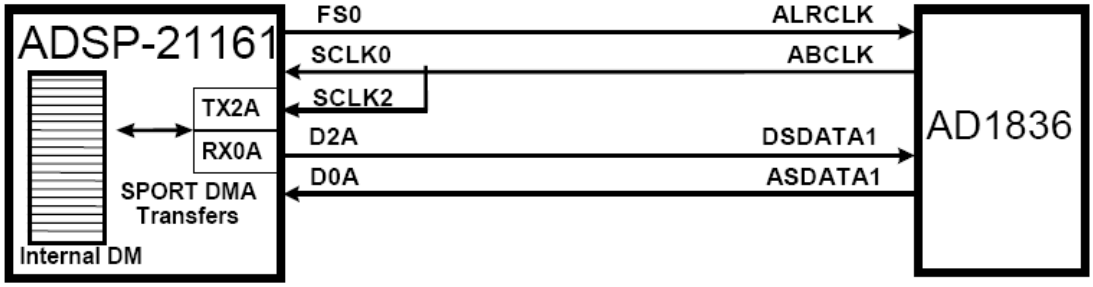


Rys. 11 Funkcjonalność pinów SPORTu w trybie wielokanałowym. Pary SPORT0-SPORT2 i SPORT1-SPORT3.

Function	SPORT0		SPORT2	
	A Chn	B Chn	A Chn	B Chn
Transmit data		X	D2A	X
Transmit clock			SCLK1	
Transmit frame sync/ word select			X (redefined as TVD2)	
Receive data	D0A	X		X
Receive clock	SCLK0			
Receive frame sync	FS0			

Tabela 1. Konfiguracja TDM portów SPORT procesora ADSP-21161

Uwaga: piny kanału B w SPORT nie funkcjonują w trybie wielokanałowym. Zarówno nadajnik jak i odbiornik mają własne zegary. Piny synchronizacji ramki FS2/FS3 stają się pinami poprawności danych wyjściowych, zaś piny FS0/FS1 służą do kontroli startu ramki wielokanałowej.



Rys.12 Typowe połączenia szeregowe między ADSP-21161 i AD1836. (przy założeniu, że zasilanie we/wy układu codeca wynosi 3.3V)

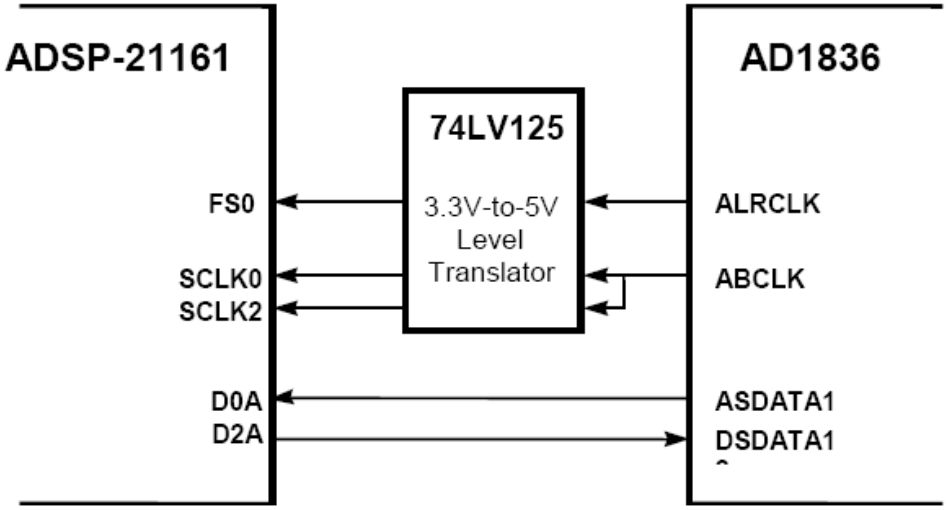
Uwaga: Linia FS2 procesora jest pinem wyjściowym w trybie wielokanałowym (TDV2 – Transmit Data Valid). Powinien pozostać nie podłączony i nie połączony z FS0 codeca. Mogłoby to spowodować kolizję na FS0 (FSTDM) i zablokować SPORT, a nawet uszkodzić pin FS0!

6. Praca ADSP-21161 przy 3,3V zasilania.

ADSP-21161 jest nową odsłoną rodziny procesorów ADSP-2116x SHARC opartej na technologii 0,18µ CMOS i pracuje na dwóch napięciach zasilania – 1,8V dla rdzenia i 3,3V dla układów we/wy. Sygnały na poziomie 5V pochodzące od codeca mogłyby zniszczyć piny procesora, operujące na 3,3V. Można tego uniknąć na dwa sposoby.

Interfejs cyfrowy codeca (cyfrowe we/wy) może opcjonalnie pracować na 3,3V. Wówczas nie jest konieczna zmiana napięcia. Jest to metoda optymalna, gdyż nie wymaga dodatkowych układów. Implementacja takiego interfejsu wymaga podłączenia w codecu pinu ODVDD do 3,3V a nie do 5V.

Drugą opcją jest zmiana poziomu wszystkich sygnałów wejściowych z codeca. Wszystkie wyjściowe sygnały SPORT są sygnałami wejściowymi dla codeca, i poziomy ich nie muszą być zmieniane, gdyż codec rozpozna 3,3V jako wysoki poziom TTL.



Rys. 13 Opcjonalny interfejs dla codeca z układami we/wy zasilanymi 5V.

W celu ułatwienia komunikacji szeregowej z codekiem, piny SPORT0 i SPORT2 są skonfigurowane jak w tabeli poniżej.

ADSP-21161N Pin:	*AD1836 Pin:*	*Driven By:*
SCLK0, SCLK2	*ABCLK*	*codec*
FS0	*FSTDM (ALRCLK)*	*codec*
FS2 [TDV2] (unconnected)	----------	----------
D0A	*ASDATA1*	*codec*
D2A	*DSDATA1*	*DSP*

Tabela 2.

7. Kanały DMA i wektory przerwań dla portów SPORT.

Istnieje 8 dedykowanych kanałów dla SPORT0/1/2/3 w procesorze ADSP-21161. Adresy IOP dla rejestrów parametrów DMA są przedstawione w tabeli poniżej, dla każdego kanału i bufora danych w SPORT. W trybie wielokanałowym jedynie kanały 0, 2, 4 i 6 są aktywowane, ponieważ w trybie tym kanał B nie funkcjonuje.

Chan	Data Buffer	Address	Description
0	RX0A/TX0A	0x0060 0x0064	Serial port 0 rx or tx; A data
1	RX0B/TX0B	0x0080 0x0084	Serial port 0 rx or tx; B data
2	RX1A/TX1A	0x0068 0x006C	Serial port 1 rx or tx; A data
3	RX1B/TX1B	0x0088 0x008C	Serial port 1 rx or tx; B data
4	RX2A/TX2A	0x0070 0x0074	Serial port 2 rx or tx; A data
5	RX2B/TX2B	0x0090 0x0094	Serial port 2 rx or tx; B data
6	RX3A/TX3A	0x0078 0x005C	Serial port 3 rx or tx; B data
7	RX3B/TX3B	0x0098 0x009C	Serial port 3 rx or tx; B data

Tabela 3. 8 kanałów DMA i bufory danych dla SPORT.

Każdy bufor portu szeregowego i pin danych ma przypisane przerwanie TX/RX DMA (patrz Tabela 4). Gdy DMA portu szeregowego jest wyłączony, przerwania występują w oparciu o słowa, w momencie gdy słowo jest przesyłane. W Tabeli 4 pokazano również priorytety przerwań ze względu na ich wzajemne położenie w tablicy wektorów przerwań. Im niższy adres wektora, tym wyższy priorytet przerwania. Należy zauważyć, że kanały A i B dla każdego SPORTu mają te same położenia przerwań. Stąd dane dla obu kanałów DMA (lub bufory danych SPORTów A i B) są przetwarzane w tym samym czasie lub warunkowo, zależnie od stanu bufora bitów statusu w rejestrach kontrolnych SPORTu.

Interrupt[1]	Function	Priority
SP0I	SPORT0 TX or RX DMA channels 0 and 1	Highest
SP1I	SPORT1 TX or RX DMA channels 2 and 3	
SP2I	SPORT2 TX or RX DMA channels 4 and 5	
SP3I	SPORT3 TX or RX DMA channels 6 and 7	
	DMA channels 8 to 13 are used for Link Ports, SPI and External Port DMA	Lowest

[1] Interrupt names are defined in the def21161.h include file supplied with the ADSP-21000 Family Visual DSP Development Software.

Tabela 4. Przerwania portu szeregowego ADSP-21161

8. Rejestry IOP związane z portem szeregowym.

Ten rozdział prezentuje listę dostępnych rejestrów IOP związanych z portem SPORT, koniecznych do konfiguracji SPORTów dla trybu wielokanałowego dal EZ-KIT Lite w celu komunikacji z codekiem przez porty SPORT0 i SPORT2. Aby zaprogramować te rejestry, korzysta się z adresów w pamięci z wykorzystaniem makr zawartych w pliku f21161.h

(udostępnionym wraz z oprogramowaniem VisualDSP w katalogu Include). Urządzenia zewnętrzne, np. drugi procesor ADSP-21161, może czytać z lub pisać do rejestrów kontrolnych w celu ustawienia operacji DMA lub uaktywnić kontrolery SPORT. Rejestry te, wypisano w Tabeli 5 poniżej. Wiele z nich wymaga programowania dla ustawienia trybu wielokanałowego. Zaznaczono je grubą czcionką.

	Register	IOP Address	Description
SPORT0	**SPCTL0**	0x1C0	SPORT0 control register
	DIV0	0x1C5	SPORT0 clock and frame sync divisor
	MR0CS0	0x1C7	SPORT0 multichannel receive select 0 (channels 31-0)
	MR0CS1	0x1C9	SPORT0 multichannel receive select 1 (channels 63-32)
	MR0CS2	0x1CB	SPORT0 multichannel receive select 2 (channels 95-64)
	MR0CS3	0x1CD	SPORT0 multichannel receive select 3 (channels 127-96)
	MR0CCS0	0x1C8	SPORT0 multichannel receive compand select 0 (channels 31-0)
	MR0CCS1	0x1CA	SPORT0 multichannel receive compand select 1 (channels 63-32)
	MR0CCS2	0x1CC	SPORT0 multichannel receive compand select 2 (channels 95-64)
	MR0CCS3	0x1CE	SPORT0 multichannel receive compand select 3 (channels 127-96)
SPORT2	**SPCTL2**	0x1D0	SPORT2 control register
	DIV2	0x1D5	SPORT2 clock and frame sync divisor
	MT2CS0	0x1D7	SPORT2 multichannel transmit select 0 (channels 31-0)
	MT2CS1	0x1D9	SPORT2 multichannel transmit select 1 (channels 63-32)
	MT2CS2	0x1DB	SPORT2 multichannel transmit select 2 (channels 95-64)
	MT0CS3	0x1DD	SPORT2 multichannel transmit select 3 (channels 127-96)
	MT2CCS0	0x1D8	SPORT2 multichannel transmit compand select 0 (channels 31-0)
	MT2CCS1	0x1DA	SPORT2 multichannel transmit compand select 1 (channels 63-32)
	MT2CCS2	0x1DC	SPORT2 multichannel transmit compand select 2 (channels 95-64)
	MT2CCS3	0x1DE	SPORT2 multichannel transmit compand select 3 (channels 127-96)
	SP02MCTL	0x1DF	SPORT 0/2 Multichannel Control Register

Tabela 5. Rejestry IOP portu szeregowego dla trybu wielokanałowego.

Wewnątrz 4-ch SPORTów procesora znajduje się 16 buforów danych zintegrowanych z 16 pinami danych. W trybie wielokanałowym dostępne bufory są tylko dla kanału A (patrz niżej). To te rejestry są właściwie używane do przesyłu danych między codekiem i kontrolerem DMA w procesorze DSP. Kontroler DMA jest używany do transferu danych z i do wewnętrznej pamięci bez ingerencji rdzenia procesora.

SPORT Data Buffers	TX0A	0x1C1	SPORT0 transmit data buffer, channel A data
	TX0B	0x1C2	SPORT0 transmit data buffer, channel B data
	RX0A	0x1C3	SPORT0 receive data buffer, channel A data
	RX0B	0x1C4	SPORT0 receive data buffer, channel B data
	TX2A	0x1D1	SPORT2 transmit data buffer, channel A data
	TX2B	0x1D2	SPORT2 transmit data buffer, channel B data
	RX2A	0x1D3	SPORT2 recieve data buffer, channel A data
	RX2B	0x1D4	SPORT2 receive data buffer, channel B data

9. Konfiguracja rejestrów IOP portów SPORT0/SPORT2 dla przetwarzania audio 48kHz.

- Długość słowa 32 bit,
- Aktywacja DMA w odbiorczym kanale A (SPORT0),
- Aktywacja DMA w nadawczym kanale A (SPORT2),
- Aktywacja łańcuchowania DMA dla SPORT0/2,
- Zewnętrzny sygnał zegarowy (SCLK0) – codec podaje zegar do procesora,
- Łańcuchowanie DMA dla nadawania i odbioru aktywne. W programie DSP deklarowane są dwa bufory – rx_buf0a[8] i tx_buf2a[0] – rezerwują 8 miejsc w

pamięci w celu zapewnienia alokacji przedziałów czasowych codeca. Rekomendowane jest łańcuchowanie DMA, gdyż obsługa przerwaniowa zużywa więcej zasobów procesora.

- Opóźnienie ramki w trybie wielokanałowym wynosi 1, czyli sygnał synchronizacji ramki następuje 1 cykl SCLK przed nadejściem pierwszego bitu słowa/przedziału czasowego (MSB) w ramce audio. Nowe ramki są oznaczane stanem wysokim na linii FSTDM na 1 cykl zegara przed pojawieniem się ramki.

```
Program_SPORT02_TDM_Registers:
      /* SPORT0 and SPORT2 are being operated in "multichannel" mode.
      This is synonymous with TDM mode which is the operating mode for the AD1836 */

      /* SPORT 0&2  Miscellaneous Control Bits Registers */
      R0 = NCH_8 | MFD1; /*Hold off on MCM enable, and no of TDM slots to 8 active channels*/
      dm(SP02MCTL) = R0; /*Multichannel Frame Delay=1, Number of Channels = 8, LB disabled*/

      /* sport0 control register set up as a receiver in MCM */
      R0 = SCHEN_A | SDEN_A | SLEN32;
      dm(SPCTL0) = R0;                /* sport 0 control register SPCTL0 = 0x000C01F0 */

      /* sport2 control register set up as a transmitter in MCM */
      R0 = SCHEN_A | SDEN_A | SLEN32;
      dm(SPCTL2) = R0;                /* sport 2 control register, SPCTL2 = 0x000C01F0 */
```

- DSP przesuwa dane z zewnętrznym zegarem 48kHz (FS0). Ponieważ zegar codeca wynosi 12,288MHz, współczynnik dzielący (256xFs) wygeneruje zegar 48kHz.

```
      /* sport 0 & 2 frame sync divide registers */
      /* External Clock and Frame Sync generated by AD1836 */
      R0 = 0x00000000;
      dm(DIV0) = R0;
      dm(DIV2) = R0;
```

- Brak kompansji.

```
      /*sport0 & sport2 receive & transmit multichannel companding enable registers*/
      R0 = 0x00000000;                /* no companding for our 8 active timeslots*/

      dm(MR0CCS0) = R0;               /* no companding on SPORT0 receive */
      dm(MR0CCS1) = R0;
      dm(MR0CCS2) = R0;
      dm(MR0CCS3) = R0;
      dm(MT2CCS0) = R0;               /* no companding on SPORT2 transmit */
      dm(MT2CCS1) = R0;
      dm(MT2CCS2) = R0;
      dm(MT2CCS3) = R0;
```

- Tryb wielokanałowy – długość = 8 słów wielokanałowych. Pozwala to na uzyskanoie 1 ramki codeca na 1 ramkę DSP.

```
      /* sport0 & sport2 receive and transmit multichannel word enable registers */
      R0 = 0x000000FF;
      dm(MR0CS0) = R0;                /* enable receive channels 0-7 */
      dm(MT2CS0) = R0;                /* enable receive channels 0-7 */

      R0 = 0x00000000;
      dm(MR0CS1) = R0;                /* clear other TDM channels in 128 timeslots*/
      dm(MT2CS2) = R0;
      dm(MT2CS3) = R0;
      dm(MT2CS1) = R0;
      dm(MT2CS2) = R0;
      dm(MT2CS3) = R0;
```

10. Rejestry DMA portów szeregowych ADSP-21161

Poniższa lista rejestrów jest dostępna w pliku defs21161.h do programowania rejestrów DMA związanych z kontrolerem DMA układów we/wy procesora. Zostanie pokazane jak rejestry te są programowane dla łańcuchowania DMA, przy czym rejestry DMA są reinicjalizowane za każdym razem, gdy zostaje wygenerowane przerwanie portu szeregowego [rejestry użyte w EZ-KIT Lite zaznaczono na niebiesko].

	DMA Register Description	DMA Register	IOP A
SPORT0 RX/TX Channel A	DMA Channel 0 Index Register	II0A	0x60
	DMA Channel 0 Modify Register	IM0A	0x61
	DMA Channel 0 Count Register	C0A	0x62
	DMA Channel 0 Chain Pointer Register	CP0A	0x63
	DMA Channel 0 General Purpose Register	GP0A	0x64
SPORT0 RX/TX Channel B	DMA Channel 1 Index Register	II0B	0x80
	DMA Channel 1 Modify Register	IM0B	0x81
	DMA Channel 1 Count Register	C0B	0x82
	DMA Channel 1 Chain Pointer Register	CP0B	0x83
	DMA Channel 1 General Purpose Register	GP0B	0x84
SPORT1 RX/TX Channel A	DMA Channel 2 Index Register	II1A	0x68
	DMA Channel 2 Modify Register	IM1A	0x69
	DMA Channel 2 Count Register	C1A	0x6A
	DMA Channel 2 Chain Pointer Register	CP1A	0x6B
	DMA Channel 2 General Purpose Register	GP1A	0x6C
SPORT1 RX/TX Channel B	DMA Channel 3 Index Register	II1B	0x88
	DMA Channel 3 Modify Register	IM1B	0x89
	DMA Channel 3 Count Register	C1B	0x8A
	DMA Channel 3 Chain Pointer Register	CP1B	0x8B
	DMA Channel 3 General Purpose Register	GP1B	0x8C
SPORT2 RX/TX Channel A	DMA Channel 4 Index Register	II2A	0x70
	DMA Channel 4 Modify Register	IM2A	0x71
	DMA Channel 4 Count Register	C2A	0x72
	DMA Channel 4 Chain Pointer Register	CP2A	0x73
	DMA Channel 4 General Purpose Register	GP2A	0x74
SPORT2 RX/TX Channel B	DMA Channel 5 Index Register	II2B	0x90
	DMA Channel 5 Modify Register	IM2B	0x91
	DMA Channel 5 Count Register	C2B	0x92
	DMA Channel 5 Chain Pointer Register	CP2B	0x93
	DMA Channel 5 General Purpose Register	GP2B	0x94
SPORT3 RX/TX Channel A	DMA Channel 6 Index Register	II3A	0x78
	DMA Channel 6 Modify Register	IM3A	0x79
	DMA Channel 6 Count Register	C3A	0x7A
	DMA Channel 6 Chain Pointer Register	CP3A	0x7B
	DMA Channel 6 General Purpose Register	GP3A	0x7C
SPORT3 RX/TX Channel B	DMA Channel 7 Index Register	II3B	0x98
	DMA Channel 7 Modify Register	IM3B	0x99
	DMA Channel 7 Count Register	C3B	0x9A
	DMA Channel 7 Chain Pointer Register	CP3B	0x9B
	DMA Channel 7 General Purpose Register	GP3B	0x9C

Tabela 6. Rejestry IOP DMA portu SPORT.

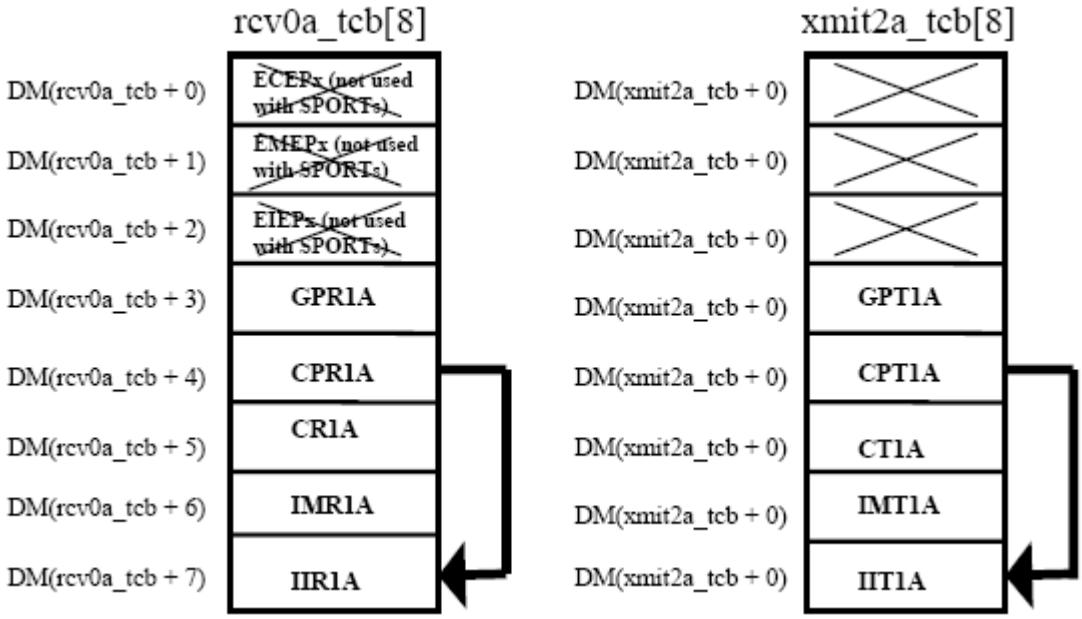
11. Konfiguracja kontrolera DMA dla łańcuchowego przesyłania danych.

Aby wydajnie przesyłać cyfrowe dane audio do i z codeca, rekomendowaną metodą jest użycie łańcuchowania DMA dla transferu danych między szyną szeregową a rdzeniem DSP. Korzyści są oczywiste. Po pierwsze, przesył DMA pozwala na efektywny transfer danych między obwodami portu szeregowego i wewnętrzną pamięcią procesora, bez udziału jego rdzenia. Po drugie, zachodzi bezpośrednia korelacja między miejscem znajdowania się słowa w nadawczych i odbiorczych buforach DMA, a aktualnym przedziałem czasowym TDM ramki audio na szynie szeregowej. Po trzecie, cały blok danych może być nadany lub odebrany przed generacją przerwania. Metoda łańcuchowania DMA jest bardziej wydajna dla procesora SHARC w przetwarzaniu danych niż przesyłanie przy użyciu przerwań, które

występują częściej i przez to powodują więcej narzutów w obsłudze danych audio. Użycie łańcuchowej metody przesyłu pozwala kontrolerowi DMA na autoinicjalizację między kolejnymi transferami DMA. Kiedy cała zawartość buforów rx_buf0a i tx_buf2a aktualnego SPORTu została wysłana/odebrana, procesor automatycznie ustawia następny proces przesyłania DMA, co jest powtarzane na każde przerwanie DMA. Dla głębszego poznania procesu łańcuchowania DMA należy sięgnąć po dokumentację sprzętową (Hardware Reference) ADSP-21161.

Rejestr wskaźnika łańcucha (CPxxx) jest używany do wskazywania na następne parametry buforów RX i TX umieszczonych w pamięci. Przesył DMA dla codeca jest inicjowany poprzez wpisanie adresu pamięci bufora DMA do rejestru CP0A dla odbioru portem SPORT0 i rejestru CP2A dla nadawania portem SPORT2. Bity SCHEN_A i SCHEN_B w rejestrze kontrolnym SPORTx uaktywniają łańcuchowanie DMA.

Aby autoinicjalizować powtarzalne transfery łańcuchowe DMA programista musi ustawić bufor w pamięci zwany Blokiem Kontroli Transferu (TCB), który będzie używany do inicjalizacji i kontynuacji procesu łańcuchowania DMA. TCB są miejscami w wewnętrznej pamięci przechowującymi informacje rejestru DMA w odpowiednim porządku. Np. rysunek poniżej pokazuje zdefiniowane TCB w pamięci dla kanału A portu SPORT1. Rejestry wskaźnika łańcucha (CP0AA i CP2A) wskazują na miejsce znajdowania się następnego zbioru parametrów TCB, mających być automatycznie pobrane przez kontroler DMA w momencie zakończenia transferu DMA, który w tym przypadku wskazuje na siebie.



Rys. 14 Bloki TCB dla łańcuchowych transferów DMA kanału A portu SPORT1 (nadawanie i odbiór)

Bloki te dla buforów nadawania i odbioru mogą być zdefiniowane w miejscu deklaracji zmiennych w kodzie programisty. Dla przykładowego kodu I2S, TCB kanału A portu SPORT1 są zdefiniowane następująco:

```
.var    rcv1a_tcb[8]  = 0, 0, 0, 0, 0, 8, 1, rx_buf0a; /* SPT0 receive tcb */
.var    xmit1a_tcb[8] = 0, 0, 0, 0, 0, 8, 1, tx_buf2a; /* SPT2 transmit tcb */
```

Należy zauważyć, że obliczenia i modyfikowane wartości mogą być inicjalizowane w deklaracji bufora, dzięki czemu są obecne po resecie procesora. Jednakże, podczas pracy, dalsza modyfikacja bufora jest wymagana do inicjalizacji procesu autobuforowania DMA.

Aby ustawić i zainicjalizować łańcuch operacji DMA w czasie pracy, program powinien postępować następująco:

1. Ustawienie TCB dla odbioru i nadawania przez SPORT. Bloki TCB są zdefiniowane w części deklaracji zmiennych w kodzie programisty. Przed ustaleniem wartości w TCB i „pozbyciu" się procesu DMA, należy się upewnić, że rejestry SPORTu są zaprogramowane z odpowiednimi bitami łańcuchowania potrzebnymi w kroku 2.
2. Wpis do rejestrów kontrolnych SPORT1/2 (SPCTL0 i SPCTL2) dla ustawienia bitów SDEN_A i/lub SDEN_B na 1, oraz bitów SCHEN_A i/lub SCHEN_B również na 1.
3. Wpisanie adresu rejestru Iixxx pierwszego TCB do rejestru CPxxx aby rozpocząć łańcuch. Kolejność powinna być następująca:
    a. Wpisanie adresu początkowego bufora DMA portu szeregowego do wewnętrznego rejestru indeksów TCB Iixxx (podstawowy adres bufora TCB+7). Pobranie adresu początkowego zdefiniowanego bufora DMA w czasie pracy i skopiowanie go do tego rejestru w TCB.
    b. Wpisanie zawartości rejestru modyfikacji DMA Imxxx do TCB (podst. adres bufora TCB+6). Ten krok może być pominięty, jeśli lokacja w buforze została zainicjalizowana w części deklaracji zmiennych w kodzie.
    c. Wpisanie zawartości rejestru licznika DMA Cxxx do TCB (podst. adres bufora TCB+5). Ten krok również można pominąć, jeśli lokacja w buforze została zainicjalizowana w części deklaracji zmiennych w kodzie.
    d. Pobranie zawartości Iixxx z bufora TCB, który został wcześniej przechowany w kroku (a), ustawienie bitu PCI z tą wartością adresu i wpisanie zmodyfikowanej wartości do lokacji wskaźnika łańcucha w TCB (bufor TCB+4).
    e. Wpisanie ręcznie tego samego bitu PCI z kroku (d) do rejestru wskaźnika łańcucha DMA (CPxxx). W tym momencie rozpoczyna się łańcuchowanie DMA.

Żądanie przerwania DMA następuje za każdym razem gdy rejestr licznika zdekrementuje się do zera.

Łańcuchowanie DMA zachodzi niezależnie dla kanałów nadawczych i odbiorczych portu szeregowego. Po wypełnieniu bufora odbioru portu SPORT (rx_buf0a) nowymi danymi, generowane jest przerwanie odbioru przez SPORT1 i odebrane dane w buforze są gotowe do przetwarzania. Kontroler DMA zainicjalizuje się samoczynnie z parametrami ustawionymi w buforze TCB i rozpocznie wypełnianie bufora odbioru DMA nowymi danymi z następnej ramki audio. Przetworzone dane są następnie są następnie umieszczone w buforze nadawczym SPORTu, skąd zostanie wysłany z pamięci do pinu DT1A portu SPORT. Po wysłaniu zawartości całego bufora, kontroler DMA ponownie zainicjalizuje się automatycznie z parametrami z TCB by wykonać kolejny transfer DMA nowych danych, które zostaną umieszczone w tym samym buforze nadawczym (tx_buf2a).

Poniżej przedstawiono instrukcje asemblera używane w sterowniku codeca, potrzebne do ustawienia buforów transmisji DMA i bloków TSB dla portów SPORT0 i SPORT2 kanału A. Wartości te są sczytywane z wewnętrznej pamięci do kontrolera DMA, po wysłaniu lub odebraniu całej zawartości bufora DMA SPORTu. Zakłada się, że SPORT0 jest skonfigurowany jako kanał odbiorczy DMA, zaś SPORT2 jako kanał nadawczy.

```
.segment /dm    dm_codec;

/* define DMA buffer sizes to match number of active TDM channels */
.var rx0a_buf[8];                       /* receive buffer (DMA)*/
.var tx2a_buf[8] =      0x00000000,     /* transmit buffer (DMA)*/
                        0x00000000,
                        0x00000000,
                        0x00000000,
                        0x00000000,
                        0x00000000,
                        0x00000000,
                        0x00000000;

/* DMA Chaining Transfer Control Block (TCB) */
/* TCB format: Ecx (length of destination buffer),

               Emx (destination buffer step size),
               Eix (destination buffer index (initialized to start address)),
               GPx ("general purpose"),
               CPx ("Chain Point register"; points to last address (IIx) of
                           next TCB to jump to upon completion of this TCB.),
               Cx  (length of source buffer),
               IMx (source buffer step size),
               IIx (source buffer index (initialized to start address))  */

.var    rcv0a_tcb[8]  = 0, 0, 0, 0, 0, 8, 1, rx0a_buf;      /* SPORT0 receive tcb */
.var    xmit2a_tcb[8] = 0, 0, 0, 0, 0, 8, 1, tx2a_buf;      /* SPORT2 transmit tcb */

.endseg;

.segment /pm    pm_code;

/*-------------------------------------------------------------------------------*/
/*          DMA Controller Programming For SPORT0 and SPORT2 primary A channels     */
/*-------------------------------------------------------------------------------*/

Program_SPORT02_DMA_Channels:

        r1 = 0x0003FFFF;                /* cpx register mask */

        /* sport2 dma control tx setup and go  */
        r0 = xmit2a_tcb + 7;            /* get DMA chaining internal mem pointer containing tx_buf address */
        r0 = r1 AND r0;                 /* mask the pointer */
                                        /*(Address will be contained in lower 18 bits (bits 17-0);
                                           Upper 13 bits will be zeroed (bits 19-31);
                                           Bit 19 is PCI bit ("Program-Controlled Interrupts") */
        r0 = BSET r0 BY 18;             /* set the pci bit */
        dm(xmit2a_tcb + 4) = r0;        /* write DMA transmit block chain pointer to TCB buffer */
        dm(CP2A) = r0;                  /* transmit block chain pointer, initiate tx0 DMA transfers */

        /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
        /* - Note: Tshift2 & TX2A will be automatically loaded with the first 2 values in the  - */
        /* - Tx buffer. The TX buffer pointer ( II2A ) will increment by 2x the modify value   - */
        /* - ( IM2A ).                                                                        - */
        /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

        /* sport0 dma control rx setup and go */
        r0 = rcv0a_tcb + 7;
        r0 = r1 AND r0;                 /* mask the pointer */
        r0 = BSET r0 BY 18;             /* set the pci bit */
        dm(rcv0a_tcb + 4) = r0;         /* write DMA receive block chain pointer to TCB buffer*/
        dm(CP0A) = r0;                  /* receive block chain pointer, initiate rx0 DMA transfers */
        RTS;

.endseg;
```

12. Zadania przedziałów czasowych TDM portu szeregowego codeca AD1836 i ich związek z buforami DMA.

Mapa przedziałów czasowych trybu wielokanałowego dla komunikacji codeca w rozszerzonym trybie TDM jest następująca:

Timeslot	DSDATA1 (D2A) Pin - Outgoing Data "Playback"	ASDATA1 (D0A) Pin - Incoming Data "Record"
0	Internal DAC0 Left Channel	Internal ADC0 Left Channel
1	Internal DAC1 Left Channel	Internal ADC1 Left Channel
2	Internal DAC2 Left Channel	External Auxiliary ADC0 Left Channel
3	External Auxiliary DAC Left Channel	External Auxiliary ADC1 Left Channel
4	Internal DAC0 Right Channel	Internal ADC0 Right Channel
5	Internal DAC1 Right Channel	Internal ADC1 Right Channel
6	Internal DAC1 Right Channel	External Auxiliary ADC0 Right Channel
7	External Auxiliary DAC Right Channel	External Auxiliary ADC1 Right Channel

Tabela 7.

rx_buf0a[8] - DSP SPORT DMA receive buffer

Slot #	*Description*	*Data Memory/DMA Buffer Direct Address Offsets*
0	Internal ADC0 Left Channel	DM(rx_buf0a + 0)
1	Internal ADC1 Left Channel	DM(rx_buf0a + 1)
2	External Auxiliary ADC0 Left Channel	DM(rx_buf0a + 2)
3	External Auxiliary ADC1 Left Channel	DM(rx_buf0a + 3)
4	Internal ADC0 Right Channel	DM(rx_buf0a + 4)
5	Internal ADC1 Right Channel	DM(rx_buf0a + 5)
6	External Auxiliary ADC0 Right Channel	DM(rx_buf0a + 6)
7	External Auxiliary ADC1 Right Channel	DM(rx_buf0a + 7)

tx_buf0a[8] - DSP SPORT DMA transmit buffer

Slot #	*Description*	*Data Memory/DMA Buffer Direct Address Offsets*
0	Internal DAC0 Left Channel	DM(tx_buf2a + 0)
1	Internal DAC1 Left Channel	DM(tx_buf2a + 1)
2	Internal DAC2 Left Channel	DM(tx_buf2a + 1)
3	External Auxiliary DAC Left Channel	DM(tx_buf2a + 2)
4	Internal DAC0 Right Channel	DM(tx_buf2a + 3)
5	Internal DAC1 Right Channel	DM(tx_buf2a + 4)
6	Internal DAC1 Right Channel	DM(tx_buf2a + 5)
7	External Auxiliary DAC Right Channel	DM(tx_buf2a + 6)

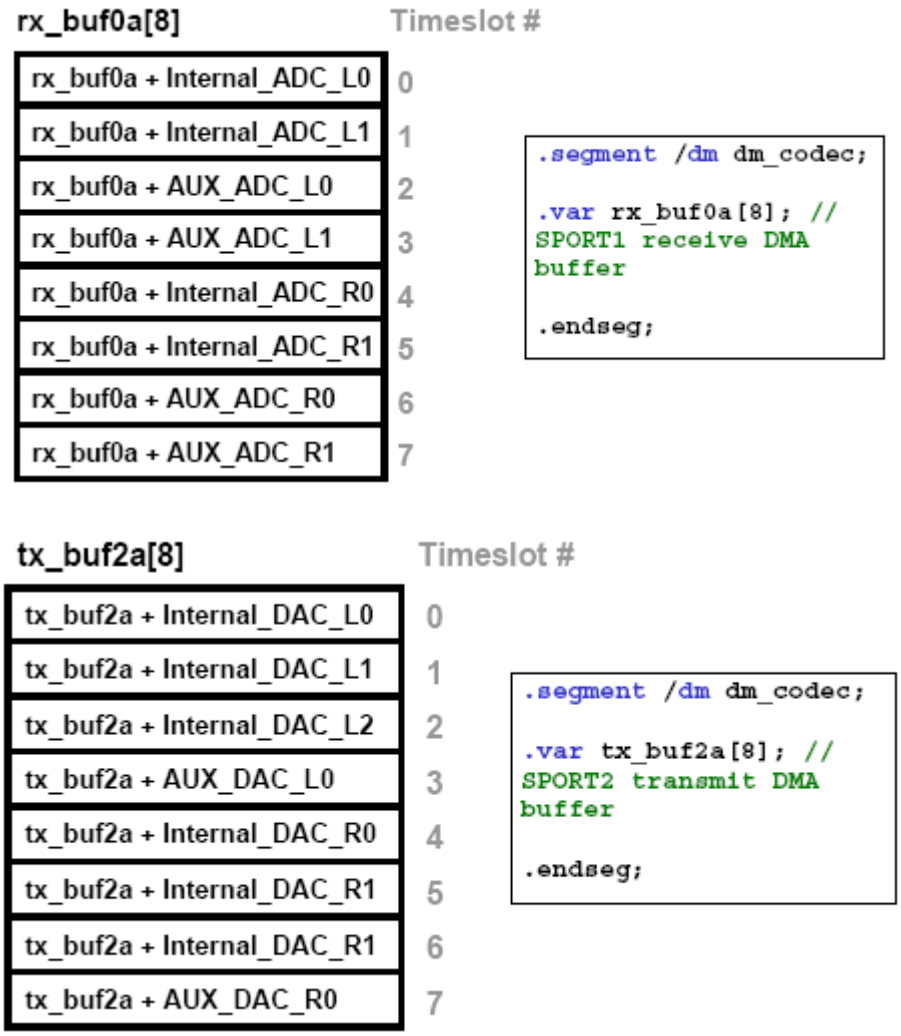
Tabela 8. Adresy buforów DMA portu SPORT0 procesora DSP dla związanych z nimi przedziałów czasowych.

Aby ułatwić odczyt ze sterownika asemblacji procesora, zdefiniowano symboliczne makrodefinicje w celu opisania każdego offsetu w buforze DMA, pokazując jego relacje z aktualnym przedziałem czasowym TDM i przetwornikami C/A i A/C codeca. Definicje te są następujące:

```
/*AD1836 TDM Timeslot Definitions */
#define         Internal_ADC_L0             0
#define         Internal_ADC_L1             1
#define         AUX_ADC_L0                  2
#define         AUX_ADC_L1                  3
#define         Internal_ADC_R0             4
#define         Internal_ADC_R1             5
#define         AUX_ADC_R0                  6
#define         AUX_ADC_R1                  7

#define         Internal_DAC_L0             0
#define         Internal_DAC_L1             1
#define         Internal_DAC_L2             2
#define         AUX_DAC_L0                  3
#define         Internal_DAC_R0             4
#define         Internal_DAC_R1             5
#define         Internal_DAC_R2             6
#define         AUX_DAC_R0                  7
```

Poniższe tabele prezentują deklarację bufora DMA portu odbiorczego SPORT0 oraz bufora portu nadawczego SPORT2, jak również symboliczne offsety dla wszystkich wewnętrznych i zewnętrznych zasobów pomocniczych codeca AD1836.



13. Programowanie portu SPI codeca AD1836.

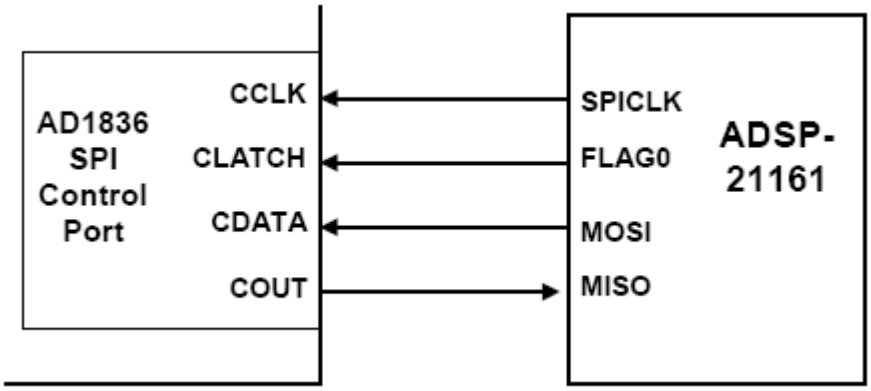
Układ AD1836 posiada 4-pinowy kontrolny port typu slave kompatybilny z SPI. Format jest podobny do formatu SPI Motoroli. Daje to rejestrowi następujące opcje dostępowe:

1. Programowanie wewnętrznych rejestrów kontrolnych dla konwerterów C/A i A/C;
2. Odczyt poziomów szczytowych sygnałów konwertera A/C poprzez detektory wartości szczytowej;
3. Poziomy wejściowe konwertera C/A mogą być programowane niezależnie, w sensie wewnętrznego tłumika dopasowywanego w 1024 liniowych krokach.

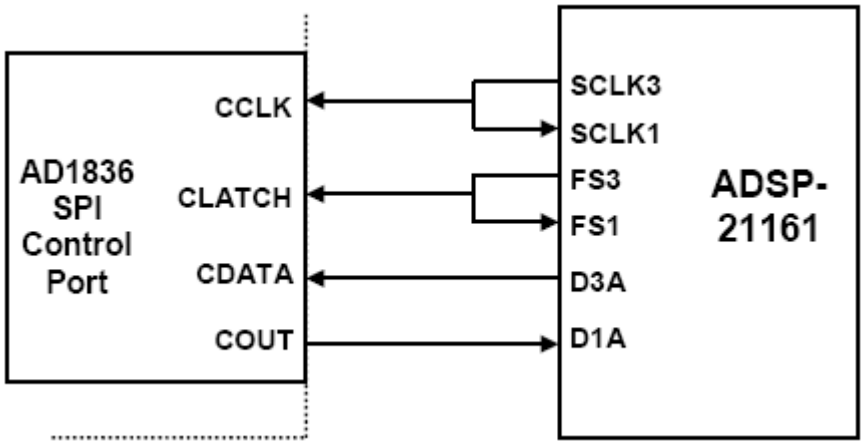
Maksymalna częstotliwość zegara dostępna przez port SPI wynosi 8MHz. Istnieją dwa sposoby programowania rejestrów codeca na płytce EZ-KIT Lite. Pierwszym jest użycie portu kompatybilnego z SPI. Drugą metodą jest emulacja SPI z użyciem portów SPORT1 i SPORT3 procesora. Jest to potrzebne do pracy wokół anomalii codeca, gdzie pin CCLK wymaga uruchomienia w trybie ciągłym do pracy wokół anomalii „dodatkowego 17-go CCLK”. Ponieważ protokół SPI używa bramkowanego zegara szeregowego, niemożliwe jest w SPI podawanie dodatkowego zegara do zatrzaskiwania danych (w SPI zegar jest bramkowany i wchodzi w stan wysoki, gdy wybór urządzenia zostanie wyłączony).

EZ-KIT Lite umożliwia korzystanie z obu metod programowania rejestrów SPI codeca. Zworka JP23 łączy piny codeca albo z portem SPI albo z portami SPORT1/SPORT3. Z załączonym JP23, porty SPORT! i SPORT3 procesora są połączone z portem SPI codeca,

zaś bez JP23, połączony jest port SPI procesora z portem SPI codeca. Poniższe schematy blokowe pokazują obie konfiguracje.



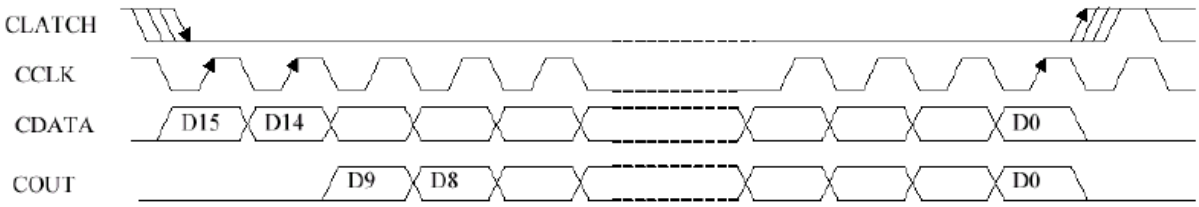
Rys. 15 AD1836 połączony z portem SPI (brak JP23)



Rys. 16 AD1836 połączony z portami SPORT1 i SPORT3 (obecnośćJP23)

W związku z anomalią SPI codeca, w referencyjnych kodach źródłowych używa się drugiej metody do interakcji portu SPI codeca. Komunikacja przez porty SPORT1 i SPORT3 pozwala na pracę SPI codeca w trybie master z ciągłym zegarem. Wówczas interfejs portu szeregowego gwarantuje dodatkowy zegar, gdy sygnał synchronizujący ramkę na porcie SPORT wejdzie w stan wysoki. Dzięki temu dane rejestru codeca nadal będą zatrzaskiwane poprawnie.

Poniższy wykres przedstawia przebiegi czasowe dla portu slave codeca. SPI codeca składa się z CLATCH (wybór urządzenia SPI), CCLK (szeregowy zegar SPI), CDATA (dane wejściowe SPI slave) i COUT (dane wyjściowe SPI slave). Format słowa SPi składa się z 16 bitów, poczynając od MSB. Dla zapisu rejestrów codeca, wpisywane dane są 16-bitowe. Dla odczytu rejestrów codeca, AD1836 podaje słowo 10-bitowe. Należy wspomnieć, że obecna wersja codeca wymaga dodatkowego CCLK po wejściu CLATCH w stan wysoki. W następnej odsłonie dodatkowy zegar nie będzie potrzebny, a dane będą zatrzaskiwane w cyklu CCLK z przesyłanym bitem D0.



Rys. 17 Przebiegi czasowe interfejsu slave SPI

W poniższej tabeli zaprezentowano format słowa SPI dla rozkazów codeca. W AD1836 znajduje się 16 wewnętrznych rejestrów. Adresy tych rejestrów są umieszczone w bitach D12 do D15 w słowie SPI. Aby zainicjować odczyt lub zapis, ustawiany jest bit D11 (1 – odczyt, 0 – zapis). Bit 10-ty jest zarezerwowany i powinien być ustawiony zawsze na 0 przez master SPI. Dane adresowane codeca mają 10-bitową długość i są złożone z bitów D0-D9.

SERIAL SPI WORD FORMAT

REGISTER ADDRESS	READ/WRITE	RESERVED	DATA FIELD
15..12	11	10	9..0
4-Bits	1=Read 0=Write	0	10-Bits

Rys. 17 Format słowa slave SPI codeca.

Codec Register Addr D15 D14 D13 D12	Read 1/ Write 0 D11	Reserved D10	Codec Register Data Function D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
0x0 (0000)		X	DAC Control 1
0x1 (0001)		X	DAC Control 2
0x2 (0010)		X	DAC Volume 0
0x3 (0011)		X	DAC Volume 1
0x4 (0100)		X	DAC Volume 2
0x5 (0101)		X	DAC Volume 3
0x6 (0110)		X	DAC Volume 4
0x7 (0111)		X	DAC Volume 5
0x8 (1000)		X	ADC 0 - Peak Level (Read Only)
0x9 (1001)		X	ADC 1 - Peak Level (Read Only)
0xA (1010)		X	ADC 2 - Peak Level (Read Only)
0xB (1011)		X	ADC 3 - Peak Level (Read Only)
0xC (1100)		X	ADC Control 1
0xD (1101)		X	ADC Control 2
0xE (1110)		X	ADC Control 3
0xF (1111)		X	*Reserved*

Tabela 9. Adresy i funkcje rejestrówAD1836

Powyższa tabela prezentuje Adresy i funkcje rejestrówAD1836. 12 programowalnych rejsetrów on-chip skategoryzowano następująco:

- 2 rejestry kontroli konwertera C/A
- 3 rejestry kontroli konwertera A/C
- 6 rejestrów kontroli poziomu sygnałów konwertera C/A
- 4 rejestry poziomu wartości szczytowych konwertera A/C

Funkcje rejestrów to m.in.:

- przełączanie między trybami I2s i TDM (2-gi rejestr kontroli A/C)
- wybór szerokości słowa danych (24, 20, 18 lub 16)
- wybór częstotliwości próbkowania (96kHz lub 48kHz)
- kontrola wzmocnienia A/C
- poziom sygnału C/A
- wyciszenie kanału A/C lub C/A

14. Konfiguracja szeregowego łącza AD1836 do trybu TDM dla kompatybilności ADI SPORT.

Rozszerzony tryb TDM pozwala na efektywną komunikację między procesorem i codekiem. Tryb ten pracuje wydajnie, gdy użyte jest autobuforowanie lub łańcuchowanie

DMA. W trybie tym wszystkie 8 przedziałów jest 32-bitowa, pozwalając prostemu interfejsowi na 32-bitowe przetwarzanie cyfrowe. DSP generuje sygnał synchronizacji ramki co 256 cykli zegara (8x32-bitowe przedziały czasowe = 256 cykli). Z SCLK pracującym na 12,288MHz, DSP generuje sygnał synchronizacji z częstotliwością 48kHz. Należy pamiętać, że w rozszerzonym trybie TDM częstotliwość próbkowania 96kHz jest niedostępna. Ay z niej skorzystać, należy użyć trybu I^2S z wykorzystaniem do 3-ech portów SPORT procesora (zależnie od ilości potrzebnych wyjść).

Domyślnie AD1836 jest w trybie I^2S. By skonfigurować codec do pracy z DSP w trybie TDM, procesor powinien zaprogramować goo pracy w tym trybie, jak tylko będzie gotowy do pracy (po resecie lub odcięciu zasilania).

```
#define         SERIAL_CONFIGURATION            0x7400
#define         ENABLE_Vfbit_SLOT1_SLOT2        0xE000

.var tx_buf[9] = ENABLE_Vfbit_SLOT1_SLOT2,    /* set valid bits for slot 0, 1, and 2 */
                 SERIAL_CONFIGURATION,        /* serial configuration register address 0x74 */
                 0xFF80,                     /* initially set to SLOT-16 mode for ADI SPORT compatibility*/
                 0x0000,                      /* stuff other slots with zeros for now */
                 0x0000,
                 0x0000,
                 0x0000,
                 0x0000,
                 0x0000;
```

15. Programowanie portów SPORT1/3 ADSP-21161 do emulacji SPI dla komunikacji z portem kompatybilnym z SPI codeca AD1836.

EZ-KIT Lite pozwala na programowanie rejestrów codeca poprzez porty SPORT1 i SPORT3. Taka emulacja pracuje dobrze przy wymaganiach dodatkowego zegara w codecu do zatrzaskiwania danych, ponieważ porty SPORT generują zegar ciągły. Codeca nie dotyczy zegar ciągły, gdy dane są zatrzaskiwane 1 cykl po tym, jak sygnał synchronizujący ramkę DSP wchodzi w stan wysoki. Aby programować rejestry codeca, SPORTy są programowane następująco:

Rejestr kontrolny SPORT3 (nadajnik):

- późny sygnał synchronizacji ramki (późny FS3 i aktywny stanem niskim FS3 emuluje operację CLATCH)
- sygnał synchronizujący ramkę zależny od danych
- wewnętrzny sygnał synchronizacji ramki
- wewnętrzny SCLK3
- 16-bitowe słowa

Rejestr kontrolny SPORT1 (odbiornik)

- późny sygnał synchronizacji ramki
- sygnał synchronizacji ramki aktywny stanem niskim
- zewnętrzny sygnał synchronizacji ramki
- zewnętrzny SCLK1 (połączony z SCLK3)
- zewnętrzny FS1 (połączony z FS3)
- 16-bitowe słowa.

Instrukcje dla DSP do konfiguracji portów SPORT1/SPORT3 do emulacji SPI w celu umożliwienia programowania rejestrów codeca przedstawiono poniżej.

```
        /* clear multichannel/miscellaneous control register for SPORT1 & SPORT3 */
        R0 = 0x0;                       dm(SP13MCTL) = R0;
        R0 = 0x0011002B;                dm(DIV3) = R0;
        R0 = 0;                         dm(DIV1) = R0;

        bit set ustat1 DDIR | SDEN_A | LAFS | LFS |  IFS | FSR | CKRE | ICLK | SLEN16 | SPEN_A;
        dm(SPCTL3) = ustat1;

        bit set ustat2 SDEN_A | LAFS | LFS | FSR | CKRE | SLEN16 | SPEN_A;
        bit clr ustat2 DDIR | IFS | ICLK;
        dm(SPCTL1) = ustat2;

        bit set imask SP1I | SP3I;     // enable SPORT1 RX and SPORT3 TX interrupts
```

16. Programowanie portu SPI Master procesora ADSP-21161 do komunikacji z portem kompatybilnym z SPI codeca AD1836.

EZ-KIT Lite pozwala na programowanie rejestrów codeca przez port SPI. Ta metoda będzie działać z nową wersją AD1836 bez dodatkowych wymagań w stosunku do zegara do zatrzaskiwania danych. Aby programować rejestry codeca przez interfejs SPI, rejestr SPICTL jest programowany następująco:

Rejestr SPICTL (urządzenie SPI Master)

- włączenie SPI
- urządzenie SPI Master
- włączenie przerwania dla nadawania SPI
- wybór urządzenia przez FLAG0
- CPHASE=1
- długość słowa SPI = 16 bit
- BER = 3,123 MHz
- pierwszy MSB
- Sign Extend

Instrukcje dla DSP do konfiguracji interfejsu SPI w celu umożliwienia programowania rejestrów codeca przedstawiono poniżej.

```
bit set LIRPTL SPITMSK;        // enable SPI TX interrupts
bit set MODE1 IRPTEN;          // allow global interrupts
bit set IMASK LPISUMI;         // unmask spi interrupts

/* configure SPI port for interface to the AD1852 */
ustat1 = dm(SPICTL);
bit set ustat1 SPIEN|SPTINT|TDMAEN|MS|FLS0|CPHASE|DF|WL16|BAUDR3|PSSE|DCPH0|SGN|GM;
bit set ustat1 CP|FLS0|FLS2|FLS3|SMLS|DMISO|OPD|PACKEN|SENDZ|RDMAEN|SPRINT;
dm(SPICTL) = ustat1;           //enable SPI port
```

17. Programowanie rejestrów kontrolnych i statusu codeca AD1836.

Addr.	*Codec Register Name*	*#define label in 21161 program*	*data bits D9:D0*	*modified by DSP?*
0x0	DAC Control 1	DAC_CONTROL1	0x000	N
0x1	DAC Control 2	DAC_CONTROL2	0x000	N
0x2	DAC Volume 0	DAC_VOLUME0	0x3FF	N
0x3	DAC Volume 1	DAC_VOLUME1	0x3FF	N
0x4	DAC Volume 2	DAC_VOLUME2	0x3FF	N
0x5	DAC Volume 3	DAC_VOLUME3	0x3FF	N
0x6	DAC Volume 4	DAC_VOLUME4	0x3FF	N
0x7	DAC Volume 5	DAC_VOLUME5	0x3FF	N
0x8	ADC 0 - Peak Level (Read Only)	ADC0_PEAK_LEVEL	0x000	N
0x9	ADC 1 - Peak Level (Read Only)	ADC1_PEAK_LEVEL	0x000	N
0xA	ADC 2 - Peak Level (Read Only)	ADC2_PEAK_LEVEL	0x000	N
0xB	ADC 3 - Peak Level (Read Only)	ADC3_PEAK_LEVEL	0x000	N
0xC	ADC Control 1	ADC_CONTROL1	0x000	N
0xD	ADC Control 2	ADC_CONTROL2	**0x380**	**Y**
0xE	ADC Control 3	ADC_CONTROL3	0x000	N
0xF	Reserved	RESERVED_REG	0x000	N

*** Registers highlighted in bold have been altered from their default states by the 21161 for the talkthru example. Other registers set by the DSP that are not highlighted but marked with a Y are set to their default reset state and are user configurable. All other registers marked with a N are not set by the DSP.*

Tabela 10. Stany rejestrów sterownika AD1836

Wszystkie używane adresowalne rejestry kontrolne codeca są ustawiane przez DSP z użyciem bufora pamięci procesora, gdzie wszystkie adresy rejestrów są przechowywane w parzystych lokacjach bufora pamięci, zaś odpowiadające im rejestry danych przechowywane

w sąsiednich lokacjach parzystych. W przypadku 21161, 11 rejestrów jest programowanych podczas inicjalizacji codeca.

W sterownikach ADI, wiele z rejestrów AD1836 nie jest modyfikowanych w stosunku do ich domyślnych wartości inicjalizowanych w momencie włączenia urządzenia. Rejestr ADC Control 2 ma wpisywaną wartość 0x380, która zmienia protokół szeregowy codeca z I^2S do rozszerzonego TDM. Rejestr ADC Control 3 jest programowany dla zapewnienia, że zegar wynosi 256 x f2 z oscylatorem 12,288 MHz, który jest obecny na wersjach 1.1 płytek 21161 EZ KIT Lite. Wartość ta może być zmieniona, jeśli użytkownik zażyczy sobie korzystać z oscylatora 24,576 MHz. Ponadto rejestry DAC Control 2 i ADC Control 1 powinny być programowane dwukrotnie po resecie codeca w celu poprawnej inicjalizacji rejestru.

Kod asemblerowy inicjalizacji bufora przedstawiono poniżej:

```
.var tx_buf3a[21] = //program register commands
                    DAC_CONTROL1 | WRITE_REG | 0x000,  // we "OR" in address, rd/wr, and register data
                    DAC_CONTROL1 | WRITE_REG | 0x000,  // for ease in reading register values
                    DAC_CONTROL2 | WRITE_REG | 0x000,  // write DAC_CTL1 twice to workaround pwdwn SPI anomaly
                    DAC_VOLUME0  | WRITE_REG | 0x3FF,
                    DAC_VOLUME1  | WRITE_REG | 0x3FF,
                    DAC_VOLUME2  | WRITE_REG | 0x3FF,
                    DAC_VOLUME3  | WRITE_REG | 0x3FF,
                    DAC_VOLUME4  | WRITE_REG | 0x3FF,
                    DAC_VOLUME5  | WRITE_REG | 0x3FF,
                    ADC_CONTROL1 | WRITE_REG | 0x000,  // write ADC_CTL1 twice to workaround pwdwn SPI anomaly
                    ADC_CONTROL1 | WRITE_REG | 0x000,
                    ADC_CONTROL3 | WRITE_REG | 0x000,  // 256*Fs Clock Mode !!!, differential PGA mode
                    ADC_CONTROL2 | WRITE_REG | 0x380,  // SOUT MODE = 110 --> TDM Mode, Master device
                    ADC_CONTROL2 | WRITE_REG | 0x380,
                    // read register commands
                    ADC0_PEAK_LEVEL | READ_REG | 0x000, // status will be in rx_buf1a[13-19] memory locations
                    ADC1_PEAK_LEVEL | READ_REG | 0x000,
                    ADC2_PEAK_LEVEL | READ_REG | 0x000,
                    ADC3_PEAK_LEVEL | READ_REG | 0x000,
                    ADC_CONTROL1    | READ_REG | 0x000,
                    ADC_CONTROL2    | READ_REG | 0x000,
                    ADC_CONTROL3    | READ_REG | 0x000;
```

18. Programowanie rejestrów AD1836 z użyciem pętli Zero Overhead.

Poniższy kod pokazuje jak wartości w buforze Init_Codec_Registers[ ] są wysyłane do odpowiednich przedziałów na szeregowej szynie TDM.

```
/* Buffer holds SPI codes to write to each of 15 internal registers on 1836 */

.VAR    Init_Codec_Registers[15] =

        DAC_CONTROL1 | WRITE_REG | 0x000,     // we "OR" in address, rd/wr, and register data
        DAC_CONTROL1 | WRITE_REG | 0x000,     // for ease in reading register values
        DAC_CONTROL2 | WRITE_REG | 0x000,     // write DAC_CTL1 twice to workaround pwdwn SPI anomaly
        DAC_VOLUME0  | WRITE_REG | 0x3FF,
        DAC_VOLUME1  | WRITE_REG | 0x3FF,
        DAC_VOLUME2  | WRITE_REG | 0x3FF,
        DAC_VOLUME3  | WRITE_REG | 0x3FF,
        DAC_VOLUME4  | WRITE_REG | 0x3FF,
        DAC_VOLUME5  | WRITE_REG | 0x3FF,
        ADC_CONTROL1 | WRITE_REG | 0x000,     // write ADC_CTL1 twice to workaround pwdwn SPI anomaly
        ADC_CONTROL1 | WRITE_REG | 0x000,
        ADC_CONTROL3 | WRITE_REG | 0x040,     // 512*Fs Clock Mode !!!, differential PGA mode
        ADC_CONTROL2 | WRITE_REG | 0x380,     // SOUT MODE = 110 --> TDM Mode, Master device
        ADC_CONTROL2 | WRITE_REG | 0x380;

_____________________________________________

// Example Core-based loop via SPORT3

Program_AD1836_Registers:
        M7 = 1;
        I7 = Init_Codec_Registers;
        L7 = 0;

        lcntr = 15, DO Set_1836_Regs until LCE;
                r13 = dm(I7,M7);
                DM(TX3A) = r13;
                idle;
Set_1836_Regs: nop;

_____________________________________________


// Example Core-based loop via SPI

Program_AD1836_Registers:
        M7 = 1;
        I7 = Init_Codec_Registers;
        L7 = 0;

        lcntr = 15, DO Set_1836_Regs until LCE;
                r13 = dm(I7,M7);
                DM(SPITX) = r13;
                idle;
Set_1836_Regs: nop;
```

Wyjaśnienie działania pętli inicjalizującej code:

- wskaźnik bufora jest najpierw ustawiany aby wskazywał na początek bufora codeca
- rejestr licznika pętli LCNTR jest ustawiany na liczbę rejestrów do zaprogramowania, w tym przypadku 11.
- zapis pamięci do DM(TX3A) lub DM(SPITX) zaprogramuje adresy rejestrów codeca
- instrukcja IDLE (bezczynności) rozkazuje procesorowi nic nie robić i czekać na przerwanie o transmisji SPORT3/SPI po wpisaniu danych do bufora nadawania portu SPORT3/SPI. Oczekiwanie na to przerwanie zagwarantuje, że wszystkie dane z bufora nadawczego zostaną wystawione na szynę szeregową, zaś użytkownik może przejść do następnego rejestru codeca z danymi w buforze inicjalizacji i umieścić słowo w kolejce do bufora nadawczego.

19. Odczyt rejestrów AD1836 dla weryfikacji i debugowania z użyciem pętli Zero Overhead.

Podczas debugowania kodu sterownika programista DSP może zechcieć zweryfikować wartości zawarte w wewnętrznych rejestrach codeca. Łatwym sposobem jest ustawienie bufora wyjściowego, w którym mogą być przechowywane wszystkie żądania odczytu rejestru po inicjalizacji codeca. Odczyt i status codeca można wykonać również

wykorzystując pętlę sprzętową. Poniższy kod jest używany do inicjacji żądań odczytu rejestrów codeca w buforze Init_Codec_Registers[ ]. Wynik żądania jest następnie umieszczany buforze wyjściowym zwanym Codec_Init_Results[]. Dla płytki 21161 EZ-KIT Lite rejestry codeca mogą być zweryfikowane przez emulator interfejsu JTAG lub debugger VDSP USB/JTAG poprzez ustawienie breakpointa po tej sekcji kodu i otwarciu okna z mapą pamięci pokazującego wartości przechowywane w buforze pamięci. Po udanym debugowaniu kodu instrukcje te można usunąć.

```
/* Verify integrity of AD1836 register states to see if communication was successful */

verify_reg_writes:
        I7 = Init_Codec_Registers;
        M7 = 1;
        I5 = Codec_Init_Results;

////////////////////////////
// SPORT1/3 version
////////////////////////////

        LCNTR = 15, Do ad1836_register_status UNTIL LCE;
                r13 = dm(I7,M7);
                DM(TX3A) = r13;
                idle;
                r3 = dm(RX1A);                  /* fetch value of requested indexed register data */
                dm(I5,1) = r3;                  /* store to results buffer */
ad1836_register_status: nop;


_____________________________________

////////////////////////////
// SPI version
////////////////////////////

        lcntr = 15, DO Set_1836_Regs until LCE;
                r13 = dm(I7,M7);
                DM(SPITX) = r13;
                idle;
                r3 = dm(SPIRX);                 /* fetch value of requested indexed register data */
                dm(I5,1) = r3;                  /* store to results buffer */
ad1836_register_status: nop;
```

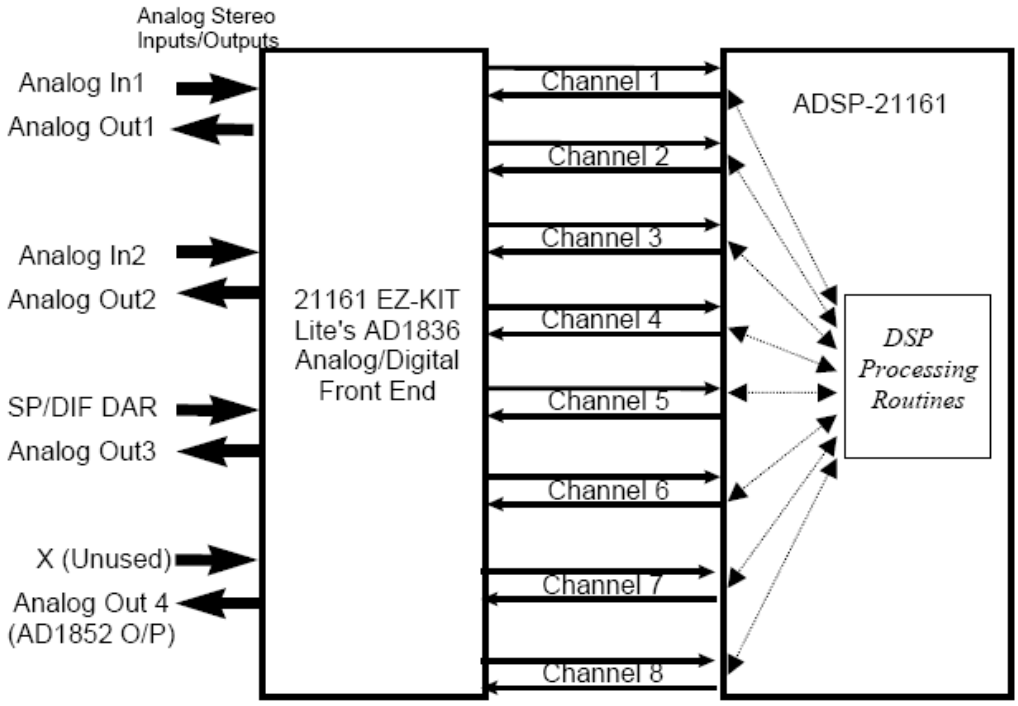
Wyjaśnienie działania pętli odczytującej (readback) rejestry codeca AD1836:

- Wskaźniki buforów I4 i I5 jako pierwsze mają wskazywać na początek bufora rejestru codeca i bufora wyników.
- Rejestr licznika pętli LCNTR jest ustawiany na ilość rejestrów codeca, które mają być odczytane
- Zapis z pamięci z TX3A lub SPITX ustawi żądanie odczytu adresu rejestru codeca zawartym w buforze Init_Codec_Registers[].
- Jedna instrukcja IDLE jest potrzebna dla poprawnego odczytu z codeca po wysłaniu żądania
- Wskaźnik I5 kopiuje adres i dane rejestru z bufora Init_Codec_Results[] za każdym żądaniem odczytu.

6. Przetwarzanie próbek dźwiękowych przez SPOTR0 RX ISR układu AD1836.

W tym rozdziale prześledzimy przykładowe instrukcje DSP aby przetwarzać dane z odbiorczego wektora przerwań SPOTR0. ADSP-21161 najczęściej przetwarza świeżo odebrane dane za pomocą przerwań SPOTR, które przekierowują kolejność wykonywania programu tak, aby przeskoczył do wektora przerwań SPORT i wykonał przerwanie. Podczas korzystania ze SPORT-ów do przetwarzania próbek dźwiękowych a AD1836 TDM, kod przerwania zawiera instrukcje, które przenoszą dane audio z danego kanału wejściowego do algorytmu przetwarzania kanałów i umieszcza rezultaty przetwarzania z powrotem do żądanej lokacji bufora tx DMA, skąd przenosi je do AD1836, gdzie dane są przetwarzane z powrotem do postaci analogowej za pomocą DACs.

Figure 26. Logical View Of AD1836/SHARC Audio System



Obrazek 26 pokazuje wysokopoziomowy widok logiczny strumieni audio, które mogą być przetworzone podłączając AD1836 do SHARC DSP. Za pomocą zawartych w AD1836 ADCs i DACs każda para szeregowych portów TDM (SPORTs 0/2, 1/3) ma możliwość przetworzenia 8 wejściowych kanałów audio i wysyłania wyjściowych strumieni audio DSP do 8 kanałów wyjściowych. Taka konfiguracja umożliwi implementację cyfrowej konsoli 8x2 kanałowej lub umożliwi uruchomienie algorytmów surround wymagających 8 kanałów audio. Będzie to za razem rozwiązanie o niskim koszcie. Jeżeli używamy zarówno SPORT0 i SPORT2 w trybie TDM, mamy możliwość podłączenia dwóch AD1836 do ADSP-21161. Rezultatem będzie sygnał audio z 16-toma kanałami we/wy.

W odniesieniu do sterownika AD1836 (dodatek A), wektor przerwań SPORT0 oraz obsługa przerwań jest wykorzystywana do przetwarzania przychodzących informacji z AD1836 przez port szeregowy. Jak opisaliśmy w rozdz. 3.4 informacje wysyłane z AD1836 są zamieniane w łańcuch DMA (np. SPORT odbiera cały blok ramki danych audio z AD1836 zanim pojawia

się przerwanie SPORT, a rejestry parametrów DMA są automatycznie przeładowywane przez procesor I/O DSP do powtórzenia transferu danych kodeka) w bufor **rx_byf0a[]** a przerwanie generowane jest kiedy bufor rx jest zapełniony nowymi danymi z wcześniej ukończonej ramki danych. Dlatego też, kiedy obsługiwane jest przerwanie RX, dane ze wszystkich aktywnych odbiorczych przedziałów czasowych (timeslots) zostały przesłane do bufora odbiorczego DMA. Kiedy odbywa się przerwanie TX, dane z bufora DMA tx zostały całkowicie przesłane do portu szeregowego we wcześniej ukończonej ramce audio. Próbki wyjściowe lewe i prawe są przesyłane do bufora transmisji DMA o nazwie **tx_buf2a[]** po to, by przesłać je ze SPORT-a. Programista ma możliwość wykonania algorytmu DSP zarówno dla przerwania transmisji DMA jak i dla przerwania odbioru DMA.

Obrazek 27 pokazuje podstawową strukturę przepływu programu SPORT RX IRQ dla przetwarzania danych audio z AD1836dla obsługiwanych asemblacji DSP i sterowników C. Diagram pokazuje przepływ próbek audio z szeregowych rejestrów buforowych do buforów DMA. Stamtąd próbki audio są kopiowane do lokacji pamięci tymczasowej (podwójnie buforowanej) aby dokonać ”talkthru” nieprzetworzonych próbek audio lub by użyć ich jako wejścia dla procedur przetwarzania audio. Dane wyjściowe są następnie kopiowane do kolejki wyjściowej do bufora transmisji DMA, skąd są później transferowane do rejestrów transmisji SPORT aby przesunąć je z portu szeregowego. Instrukcje przerwania kodeka AD1836 mogą być wykonane z przerwaniem transmisji bądź odbioru.

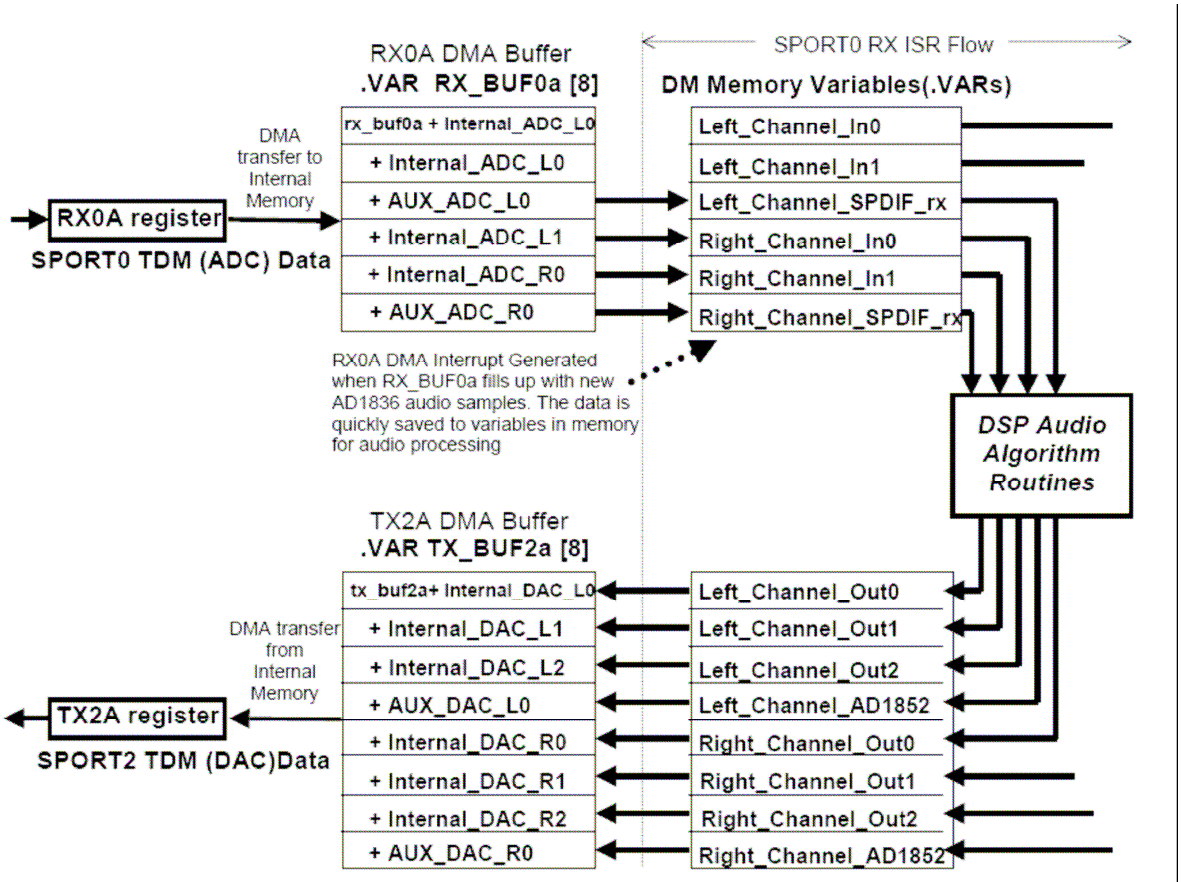


Fig 27. SPORT 0 RX Interrupt Service Routine Data Flow Structure for AD1836 Audio Processing

SPORT0 RX Interrupt Service Routine Workflow

1) Get new audio samples received from AD1836 via the SPORT receive DMA buffer rx_buf0a[ ] .
2) Save new audio samples to Audio Variables in Memory for DSP Processing
6) Run Desired DSP Algorithm
4) Retrieve processed audio samples from Audio Variables in Memory
7) Copy DSP Algorithm Results to AD1836 DACs via the SPORT DMA buffer tx_buf2a[ ] .

Proszę zwrócić uwagę, że kod sterownika odwołania AD1836 kopiuje wszystkie dane przychodzące z bufora transmisji DMA SPORT do tymczasowych lokacji pamięci DSP lub zmiennych. Kiedy dane audio zostaną przetworzone przez ADSP-21161, wyniki są kopiowane do tymczasowych zmiennych wyjściowych w pamięci DSP, gdzie zostaną odczytane przez procedurę obsługi przerwania SPORT0 i skopiowane do wyjściowego bufora transmisji DMA. Proszę zwrócić uwagę, że wszystkie zestawy EZ-KIT i dema C dostarczane przez Analog Devices korzystają z tego samego standardu nazw dla wejściowych i wyjściowych zmiennych audio. Tabela 11 pokazuje definicje tych zmiennych , zarówno w asemblerze i C oraz jak są one powiązane z buforami transmisji i odbioru DMA:

TABLE11. DMA Buffer relationship to Data Memory Audio Variables

Timeslot #	SPORT DMA Buffer Timeslot Data	=	Temp Audio Variable Equivalent
0	DM(rx_buf0a + Internal_ADC_L0)	←→	DM(Left_Channel_In0)
1	DM(rx_buf0a + Internal_ADC_L1)	←→	DM(Left_Channel_In1)
2	DM(rx_buf0a + AUX_ADC_L0)	←→	DM(Left_Channel_SPDIF_rx)
3	DM(rx_buf0a + AUX_ADC_L1)	←→	*X (not used in EZ-KIT Lite)*
4	DM(rx_buf0a + Internal_ADC_R0)	←→	DM(Right_Channel_In0)
5	DM(rx_buf0a + Internal_ADC_R1)	←→	DM(Right_Channel_In1)
6	DM(rx_buf0a + AUX_ADC_R0)	←→	DM(Left_Channel_SPDIF_rx)
7	DM(rx_buf0a + AUX_ADC_R0)	←→	*X (not used in EZ-KIT Lite)*
0	DM(tx_buf2a + Internal_DAC_L0)	←→	DM(Left_Channel_Out0)
1	DM(tx_buf2a + Internal_DAC_L1)	←→	DM(Left_Channel_Out1)
2	DM(tx_buf2a + Internal_DAC_L2)	←→	DM(Left_Channel_Out2)
3	DM(tx_buf2a + AUX_DAC_L0)	←→	DM(Left_Channel_AD1852)
4	DM(tx_buf2a + Internal_DAC_R0)	←→	DM(Right_Channel_Out0)
5	DM(tx_buf2a + Internal_DAC_R1)	←→	DM(Right_Channel_Out1)
6	DM(tx_buf2a + Internal_DAC_R2)	←→	DM(Right_Channel_Out2)
7	DM(tx_buf2a + AUX_DAC_R0)	←→	DM(Right_Channel_AD1852)

Assembly Variable Declarations

```
.segment /dm    dm_codec;

/* AD1836 stereo-channel data holders - used for DSP processing of audio data received from codec */
.VAR    Left_Channel_In0;       /* Input values from AD1836 ADCs */
.VAR    Left_Channel_In1;
.VAR    Right_Channel_In0;
.VAR    Right_Channel_In1;
.VAR    Left_Channel_SPDIF_rx;
.VAR    Right_Channel_SPDIF_rx;

.VAR    Left_Channel_Out0;      /* Output values for AD1836 DACs */
.VAR    Left_Channel_Out1;
.VAR    Left_Channel_Out2;
.VAR    Right_Channel_Out0;
.VAR    Right_Channel_Out1;
.VAR    Right_Channel_Out2;
.VAR    Left_Channel_AD1852;
.VAR    Right_Channel_AD1852;

.VAR    Left_Channel;           /* can use for intermediate results to next filter stage */
.VAR    Right_Channel;          /* can use for intermediate results to next filter stage */

.endseg;
```

C-Style Variable Declarations

```
C-Callable Assembly Variable Declarations in ADDS_21161_EZKIT.ASM

/* AD1836 stereo-channel data holders - used for DSP processing of audio data received from codec */

// input channels
.var    _Left_Channel_In0;              /* Input values from the 2 AD1836 internal stereo ADCs */
.var    _Left_Channel_In1;              /* 1/8th inch stereo jack connected to internal stereo ADC1 */
.var    _Right_Channel_In0;
.var    _Right_Channel_In1;
.var    _Left_Channel_SPDIF_rx;         /* Input values from the DAR CS8414 */
.var    _Right_Channel_SPDIF_rx;

//output channels
.var    _Left_Channel_Out0;             /* Output values for the 3 AD1836 internal stereo DACs */
.var    _Left_Channel_Out1;             /* Left and Right Channel 0 DACs go to headphone jack */
.var    _Left_Channel_Out2;
.var    _Right_Channel_Out0;
.var    _Right_Channel_Out1;
.var    _Right_Channel_Out2;
.var    _Left_Channel_AD1852;           /* Output values for AD1852 stereo DAC */
.var    _Right_Channel_AD1852;

Equivalent External C Definitions in ADDS_21161_EZKIT.H

// input channels
extern  int     Left_Channel_In0;       /* Input values from the 2 AD1836 internal stereo ADCs */
extern  int     Left_Channel_In1;       /* 1/8th inch stereo jack connected to internal stereo ADC1 */
extern  int     Right_Channel_In0;
extern  int     Right_Channel_In1;
extern  int     Left_Channel_SPDIF_rx; /* Input values from the DAR CS8414 */
extern  int     Right_Channel_SPDIF_rx;

//output channels
extern  int     Left_Channel_Out0;      /* Output values for the 3 AD1836 internal stereo DACs */
extern  int     Left_Channel_Out1;      /* Left and Right Channel 0 DACs go to headphone jack */
extern  int     Left_Channel_Out2;
extern  int     Right_Channel_Out0;
extern  int     Right_Channel_Out1;
extern  int     Right_Channel_Out2;
extern  int     Left_Channel_AD1852;    /* Output values for AD1852 stereo DAC */
extern  int     Right_Channel_AD1852;
```

6.1 ADSP-21161 SPORT DMA & Wielokanałowy AD1836 – Uwagi Czasowe

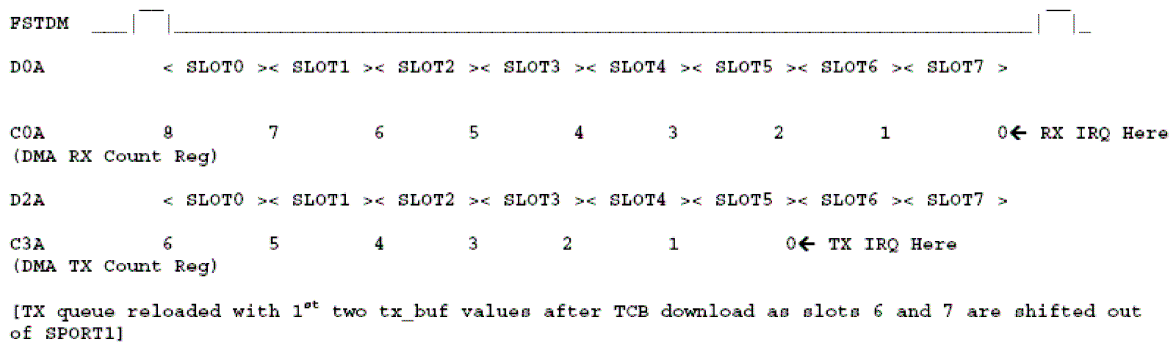
W zależności czy dane przetwarzane są z przerwania transmisji SPORT2 czy przerwania odbioru SPORT0, programista powinien być świadom różnic czasowych przerwań szeregowych RX/TX. Jeśli przetwarzamy ze SPORT2, DSP będzie przetwarzał przychodzące dane rx w aktualnej ramce dla przedziałów czasowych od 0 do 5. Jednakże, przedziały czasowe 6 i 7 aktualnej ramki audio mogą nie zostać przetworzone do czasu otrzymania ramki audio lub następnego przerwania transmisji. Podobnie, jeśli przetwarzamy dane ze SPORT0, dane tx DMA dla przedziałów czasowych od 0 do 1 zostały już przeniesione z

buforów transmisji DMA do kolejki FIFO SPORT tx. Dlatego nowo przetwarzane dane dla przedziałów czasowych od 0 do 1 nie będą przenoszone do czasu następnej ramki audio.

Powodem tej różnicy jest to, iż kiedy używamy łańcuchowania SPORT TX i RX DMA w trybie TDM, przerwania DMA są zawsze od siebie oddalone o co najmniej dwa przedziały czasowe (obrazek 28). Jest tak, ponieważ Transmit TCB początkowo umieszcza pierwsze 2 słowa z bufora DMA tx w rejestrach bufora SPORT1 TX. To automatycznie zmniejsza licznik rejestru transmisji DMA o 2. Po potwierdzeniu przerwania TX „złańcuchowanego" DMA, dane dla kanałów 6 i 7 nie zostały jeszcze wysłane do wewnętrznej pamięci.

Dlatego, zanim przedział czasowy 0 zacznie nadawać/odbierać, **RX DMA Count = 8** podczas gdy **TX DMA Count = 6** (biorąc pod uwagę, że zadeklarowaliśmy ośmiosłowowe bufory TX DMA i RX DMA z ośmioma aktywnymi przedziałami czasowymi na porcie szeregowym). Przerwanie transmisji pojawia się, gdy TX DMA Count = 0 i to żądanie przerwania pojawia się natychmiast na drugi okres zegara DSP po tym, jak LSB 5-tego przedziału czasowego zostanie przetransmitowany. Podczas gdy to przerwanie transmisji jest generowane, dane dla przedziału czasowego prawokanałowego DAC3 układu AD1836 dla kanału 7 są w kolejce w rejestrze TX2A i czekają by zostać przeniesione po tym, jak dane z 6-tego przedziału czasowego zostaną przesunięte ze SPORT-a. Jak oba przerwania (transmisji i odbioru) zostaną zatrzaśnięte w aktualnej ramce (po przedziale czasowym 5(tx) lub 7(tx)), TCB zostanie przeładowany, a po nim transfery wewnętrznej pamięci DMA.

Figure 28. AD1836/SPORT Timeslot, DMA Count and RX & TX Interrupt Timing Relationships



(1 / 12.288 MHz SCLK) x (32-bits/timeslot) x (2 timeslots) = 5.208 microseconds

1 / 100 MHz Instruction Execution = 10 nanoseconds per instruction

5.208 microseconds / 10 nanoseconds = 156.25 = 521 DSP CCLK cycles TX/RX DMA IRQ difference

Ważne, by znać te różnice czasowe w DMA. Jednakże nie są one katastrofalne dla projektanta systemu DSP. Ponieważ AD 1836 nie posiada rejestru kontrolnego ani znacznika prawidłowości informacji w przedziałach czasowych tak, jak mają kodeki AC-97 (które są bardzo wrażliwe na różnice czasowe DMA), sprawy opóźnienia ramki audio układu AD1836 nie są problemem, ponieważ użytkownik nie potrafi zauważyć opóźnienia audio jednej próbki 48 KHz między wejściem z AD1836 do SPORT0 oraz wyjściem z SPORT2 do AD1836.

6.2 Wielokanałowe metody implementacji DMA i ISR dla przetwarzania danych 48kHz.

Teraz, kiedy zbadaliśmy w rozdz. 6.1 różnice czasowe przerwań SPORT TX i RX pomiędzy kanałami transmisji i odbioru, omówimy metody implementacji przerwania odbioru SPORT0 do przetworzenia strumieni audio do i z AD1836.
W niektórych aplikacjach, użytkownik może nie chcieć przetwarzać danych gdzieindziej. Np. w aplikacjach opartych na C, procedury C-uruchomieniowe mogą być umieszczone w głównej pętli „while" programu, czekając na przerwanie SPORT. Odpowiedzialność procedury obsługi przerwania kodeka polegałaby na odbiorze i transmisji danych kodeka, podczas gdy przetwarzanie danych odbywałoby się gdzieindziej. Np. przykładowe dema ADSP-21161 EZ-KIT-u używają schematu podwójnego buforowania, który umożliwia użytkownikowi kopiowanie danych do bufora tymczasowego tak, że podczas gdy bufory DMA są aktualnie zapełniane, użytkownik przetwarza dane z buforów alternatywnych w tle. Po tym jak dane audio zostaną przetworzone, informacja jest kopiowana do bufora transmisji użytkownika.

Aby przygotować dane DAC do transmisji do następnej ramki, instrukcje SPORT ISR procesora DSP powinny po prostu dołączyć transfery danych DM do odpowiednich lokacji byfora transmisji SPORT DMA, który z kolej jest transferowany z portu szeregowego do następnego zapenienia synchronizacji ramki TDM. Natępnie procedura przerwania SPORT procesora DSP wykonuje instrukcje aby upewnić się, że umieści ona przetwarzane dane w przedziałach czasowych lewego (slot0, slot1, slot2, slot3) i prawego (slot4, slot5, slot6, slot7) kanału.

Ta metoda przetwarzania pojedynczej próbki jest podobna do kolejki FIFO, w której zawsze będziemy transmitowali nowo przetwarzaną próbkę do DACs w następnej ramce audio za każdym razem, gdy dostajemy nową próbkę ADC i ją przetwarzamy.

Spójrzmy na Assembly Language Instructions układu ADSP-21161, które zawarte są w Procedurze Obsługi Przerwania SPORT0 kodeka 21161 (w dodatku A). Pokazują one jak przetwarzana jest próbka audio.

1) Poniższe instrukcje demonstrują jak skopiować nowe dane przetwornika ADC układu AD1836 dla wszystkich przychodzących przedziałów czasowych i zachować aktualne dane lewego i prawego kanału do przetworzenia („podwójnie buforuje" to dane wejściowe tak, że nie są one nadpisane przez aktualne operacje odbioru SPORT DMA podczas wykonywania większych algorytmów):

```
Process_AD1836_Audio_Samples:
        /* get AD1836 left channel input samples, save to data holders for processing */
        r0 = dm(rx0a_buf + Internal_ADC_L0);  dm(Left_Channel_In0) = r0;
        r0 = dm(rx0a_buf + Internal_ADC_L1);  dm(Left_Channel_In1) = r0;
        r0 = dm(rx0a_buf + AUX_ADC_L0);       dm(Left_Channel_SPDIF_rx) = r r0;

        /* get AD1836 right channel input samples, save to data holders for processing */
        r0 = dm(rx0a_buf + Internal_ADC_R0);  dm(Right_Channel_In0) = r0;
        r0 = dm(rx0a_buf + Internal_ADC_R1);  dm(Right_Channel_In1) = r0;
        r0 = dm(rx0a_buf + AUX_ADC_R0);       dm(Right_Channel_SPDIF_rx) = r0;
```

2) Natępnie wywołujemy nasz algorytm DSP:

```
do_audio_processing:
        call (pc, process_audio);
```

3) Po przetworzeniu danych wejściowych ADC, wysyłamy przetworzone rezultaty do przetwornika DAC układu AD1836 w następnej ramce audio. Aby to ukończyć, po prostu kopiujemy rezultaty z naszych wejściowych zmiennych audio i umieszczamy je w kolejce SPORT tx DMA w tx_buf2[].

```
        /* ---- DSP processing is finished, now playback results to AD1836 ---- */

playback_AD1836_left_DACs:                  /* output processed left ch audio samples to AD1836 */
        r0 = dm(Left_Channel_Out0);         dm(tx2a_buf + Internal_DAC_L0) = r0;
        r1 = dm(Left_Channel_Out1);         dm(tx2a_buf + Internal_DAC_L1) = r1;
        r2 = dm(Left_Channel_Out2);         dm(tx2a_buf + Internal_DAC_L2) = r2;
        r3 = dm(Left_Channel_AD1852);       dm(tx2a_buf + AUX_DAC_L0) = r3;

playback_AD1836_right_DACs:                 /* output processed right ch audio samples to AD1836 */
        r0 = dm(Right_Channel_Out0);        dm(tx2a_buf + Internal_DAC_R0) = r0;
        r1 = dm(Right_Channel_Out1);        dm(tx2a_buf + Internal_DAC_R1) = r1;
        r2 = dm(Right_Channel_Out2);        dm(tx2a_buf + Internal_DAC_R2) = r2;
        r3 = dm(Right_Channel_AD1852);      dm(tx2a_buf + AUX_DAC_R0) = r3;
```

6.3 Przetwarzanie danych 24-bitowych w formacie cząstkowym 1.31 lub 32-bitowym formacie zmiennoprzecinkowym IEEE.

Dane, które są otrzymane bądź transmitowane w SPORT1 ISR są binarne, w kodzie z uzupełnieniem do 2. DSP tłumaczy dane na format cząstkowy, gdzie wszystkie liczby są pomiędzy –1 i 0.9999999. Ponadto port szeregowy umieszcza dane w pamięci wewnętrznej w bitach od D0 do D15. Aby przetworzyć dane cząstkowe w formacie 1.31 układ Ad1836 wyrównuje do lewej strony dane 24-bitowe do górnych 32-bitów słowa danych. Sprawa staje się prostsza i można wtedy skorzystać ze stałoprzecinkowego mnożnika/akumulatora cząstkowego trybu 1.31. Umożliwia to również prostą zamianę cząstkowego trybu 1.31 do formatów zmiennoprzecinkowych. Gwarantuje to także, że błędy kwantyzacji powodowane obliczeniami pozostaną dużo poniżej rezultatu 24-bitowego i w ten sposób poniżej 105dB poziomu szumu przetwornika DAC układu AD1836. Po przetworzeniu danych, cząstkowy rezultat w trybie 1.31 jest przesyłany do AD1836, który obcina słowo 32-bitowe do postaci 24-bitowej przed konwersją DAC. Poniżej są przykładowe instrukcje, które demonstrują przesunięcie danych przed i po przetworzeniu danych w lewym kanale Master układu AD1836.

32-bit Fixed Point Processing

```
r1 = dm(rx_buf0a + Internal_ADC_L0);         /* get AD1836 left channel ADC0 input sample */
dm(Left_Channel_In0)=r1;                     /* save to data holder for processing */

/*   Process data here, data is processed in 1.31 format */
[Call Fixed_Point_Algorithm]

r15 = dm(Left_Channel_Out0);                 /* get channel 1 output result */
dm(tx_buf2a + Internal_DAC_L0) = r15;        /* output left result to AD1836 left channel DAC0 */
```

32-bit Floating Point Processing

```
To convert between our assumed 1.31 fractional number and IEEE floating point math, here are some example
assembly instructions.  This assumes that our AD1836 data has already been converted to floating point
format, as shown above:

r1 = -31;     <-- scale the sample to the range of +/-1.0
r0 = DM(Left_Channel_In0);
f0 = float r0 by r1;

[Call Floating_Point_Algorithm]

r1 = 31;      <-- scale the result back up to MSBs
r8 = fix f8 by r1;
DM(Left_Channel_Out0) = r8;
```

7. Opis sterownika DSP ADSP-21161/AD1836.

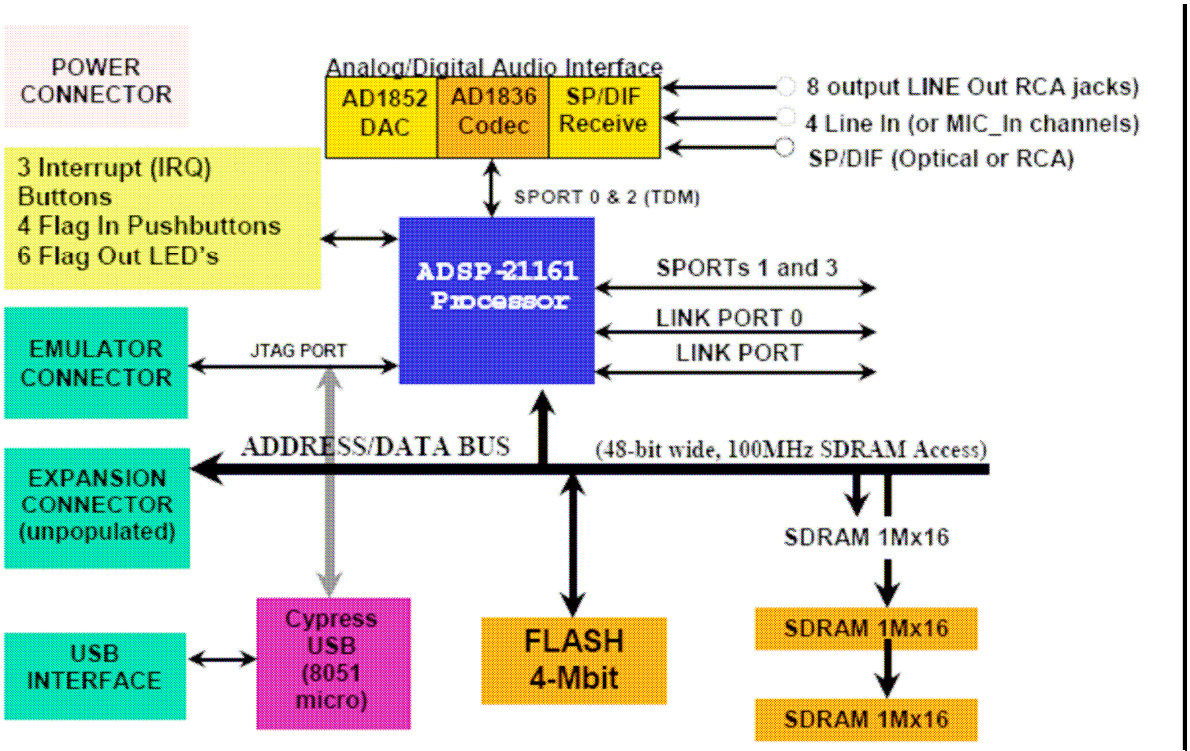


Figure 29. 21161 EZ-KIT Lite Audio Development System

Źródła kodów procesora DSP dla inicjalizacji układu AD1836 i przetwarzania audio (zawarte w dodatku A) mogą być głównym punktem startowym aby rozwinąć kod układu AD1836. Przykładowy program ADSP-21161 inicjalizuje szeregowy port DSP by komunikować się z portem szeregowym AD1836 i w następnej kolejności wykonać funkcję „talkthru" danych audio do i z AD1836. Żadne przetwarzanie DSP nie jest dokonywane po inicjalizacji. Jedyną wykonywaną operacją jest przechwytywanie danych z przetworników ADC układy AD1836 i wyprowadzanie tych samych danych z przetworników ADC tego układu.

Sterowniki układu ADSP-21161/AD1836 EZ-KIT Lite w dodatku A są zorganizowane wg następującej kolejności:

1. Procedura inicjalizacji systemu 21161 EZ-KIT
2. Procedura inicjalizacji AD1836 (dla wykonania przerwania SPORT0 RX)
3. Resetowanie AD1826 przez DSP Slave SPI Control
4. Procedura czyszczenia rejestrów SPORT
5. Procedura obsługi przerwania SPORT1 RX układu ADSP-21161...używana do przetworzenia audio
6. tablica wektorów przerwań układu ADSP-21161
7. Plik programu Visual DSP z opisem linkera

Przykładowy program dla ADSP-21161 DSP wykonuje następujące operacje do połączenia z AD1836 i przetworzenia danych audio:

1. Inicjalizacja systemu DSP (liczniki, Dagi, flag pins...)
2. Inicjalizacja portu szeregowego rejestrów 0 i 2

3. Programowanie kontrolera DMA dla łańcuchowania DMA portów szeregowych 0/2
4. Włączenie portu szeregowego 0/2 i umożliwienie SPORT0 aby odebrał przerwanie
5. Reset/Power Cycle the AD1836
6. Programowanie wybranych rejestrów układu AD1836
7. Początek przetwarzania danych audio.

APPENDIX A:
Assembly Source Code Listing for 21161 EZ-KIT Lite Audio Driver (Visual DSP Project Files)
21161 EZ-KIT System Initialization Routine

```
/*** INIT_21161_EZKIT.ASM ************************************************
* *
* ADSP-21161 EZ-KIT Initialization and Main Program Shell *
* Developed using the ADSP-21161 EZ-KIT LIte Evaluation Platform *
* *
* *
* John Tomarakos *
* ADI DSP Applications Group *
* Revision 1.0 *
* 1/30/01 *
* *
*****************************************************************************/
/* ADSP-21161 System Register bit definitions */
#include "def21161.h"
.GLOBAL _main;
.GLOBAL Init_DSP;
//.EXTERN init_21161_SDRAM_controller;
//.EXTERN Program_SPORT02_TDM_Registers;
//.EXTERN Program_AD1836_regs_via_SPI;
//.EXTERN Program_SPORT02_DMA_Channels;
//.EXTERN AD1836_Codec_Initialization;
//.EXTERN Init_AD1852_DACs;
//.EXTERN Reg_SPI_Code_init;
//.EXTERN Clear_All_SPT_Regs;
//.EXTERN user_code_init;
/*---------------------------------------------------------------------------*/
.section /pm pm_code;
_main:
/* This may be required for disabling SPORT config for EZLAB debugger */
CALL Clear_All_SPT_Regs; /* Clear and Reset SPORTs and DMAs */
call init_21161_SDRAM_controller; /* Initialize External Memory */
call Program_AD1836_regs_via_SPI; /* Use SPORTs 1&3 to talk to AD1836 SPI interface */
/* and program AD1836 Registers*/
call Init_AD1852_DACs;
call Program_SPORT02_TDM_Registers; /* Initialize SPORTs for codec communications */
call Program_SPORT02_DMA_Channels; /* Start Serial Port 1 tx and rx DMA Transfers */
call user_code_init; /* initialize user buffers */
IRPTL = 0x00000000; /* clear pending interrupts */
bit set imask SP0I|SP2I; /* start audio processing, enable SPORT1 tx & rx int */
bit set imask IRQ0I | IRQ1I | IRQ2I; /* irq0, irq1 and irq2 enabled */
ustat2 = dm(SP02MCTL);
bit set ustat2 MCE;
dm(SP02MCTL) = ustat2;
call Blink_LEDs_Test; /* Are We Alive? */
wait_forever: /* wait forever loop */
idle;
jump wait_forever;
/*---------------------------------------------------------------------------------------*/
/* Note: This routine is first called at the Reset Vector in the Interrupt Vector Table */
/*---------------------------------------------------------------------------------------*/
Init_DSP:
/* *** Enable circular buffering in MODE1 Register for revision 0.x silicon.
Important when porting 2106x code!!! */
bit set MODE1 CBUFEN;
/* Setup hardware interrupts, FLAG LEDs and pushbutton */
ustat2=0x00000000;
/* flags 4-9 are outputs for LEDs, turn on all LEDs*/
bit set ustat2 FLG9O|FLG8O|FLG7O|FLG6O|FLG5O|FLG4O;
bit set ustat2 FLG9|FLG8|FLG7|FLG6|FLG5|FLG4;
dm(IOFLAG)=ustat2;
bit clr MODE2 FLG0O | FLG1O | FLG2O | FLG3O; /* flag 0-3 are inputs from pushbutton switches
*/
IMASK = 0x0;
LIRPTL = 0x0;
IRPTL = 0x00000000; /* clear pending interrupts */
bit set mode2 IRQ2E | IRQ0E | IRQ1E; /* irqx edge sensitive */
bit set mode1 IRPTEN | NESTM; /* enable global interrupts, nesting */
bit set imask IRQ0I | IRQ1I | IRQ2I; /* irq0, irq1 and irq2 enabled */
L0 = 0;
```

```
L1 = 0;
L2 = 0;
L3 = 0;
L4 = 0;
L5 = 0;
L6 = 0;
L7 = 0;
L8 = 0;
L9 = 0;
L10 = 0;
L11 = 0;
L12 = 0;
L13 = 0;
L14 = 0;
L15 = 0;
rts;
Blink_LEDs_Test:
/* Setup FLAG 4-11 outputs */
ustat2=dm(IOFLAG);
bit set ustat2 FLG9O|FLG8O|FLG7O|FLG6O|FLG5O|FLG4O;
bit clr ustat2 FLG9|FLG8|FLG7|FLG6|FLG5|FLG4; /* clear flags to start*/
dm(IOFLAG)=ustat2;
/* Blink flags 5 times (twice per second) */
lcntr=10, do blink_loop until lce;
lcntr=6250000;
do delay until lce;
delay: nop;
bit tgl ustat2 FLG9|FLG8|FLG7|FLG6|FLG5|FLG4;
blink_loop: dm(IOFLAG)=ustat2;
rts;
```

AD1836 Initialization Routine

```
/*** SPORTs1&3_SPI_Emulation.ASM ***************************************
* *
* AD1836/ADSP-21161 SPI Code Register Initialization via SPI Emulation *
* *
* John Tomarakos *
* ADI DSP Applications Group *
* Revision 2.0 *
* 03/05/01 *
* *
*******************************************************************************/
/* ADSP-21161 System Register bit definitions */
/* refer to latest DEF21161.H file for SPORT bitfield definitions */
#include "def21161.h"
.GLOBAL Program_AD1836_regs_via_SPI;
.GLOBAL Count_SPORT1_RX_IRQs;
.GLOBAL Count_SPORT3_TX_IRQs;
//.EXTERN Wait_Approx_999us;
.GLOBAL Wait_Approx_1500ms;
.GLOBAL Wait_Approx_167ms;
// AD1836 codec SPI control/status register definitions
#define READ_REG 0x0800
#define WRITE_REG 0x0000
#define DAC_CONTROL1 0x0000
#define DAC_CONTROL2 0x1000
#define DAC_VOLUME0 0x2000
#define DAC_VOLUME1 0x3000
#define DAC_VOLUME2 0x4000
#define DAC_VOLUME3 0x5000
#define DAC_VOLUME4 0x6000
#define DAC_VOLUME5 0x7000
#define ADC0_PEAK_LEVEL 0x8000
#define ADC1_PEAK_LEVEL 0x9000
#define ADC2_PEAK_LEVEL 0xA000
#define ADC3_PEAK_LEVEL 0xB000
#define ADC_CONTROL1 0xC000
#define ADC_CONTROL2 0xD000
#define ADC_CONTROL3 0xE000
#define RESERVED_REG 0xF000
.section/dm dm_data;
/* Powerdown ADCs and DACs twice to get around AD1836 SPI/powerdown anomaly */
.var powerdown_AD1836[4] = DAC_CONTROL1 | WRITE_REG | 0x004,
```

```
DAC_CONTROL1 | WRITE_REG | 0x004,
ADC_CONTROL1 | WRITE_REG | 0x080,
ADC_CONTROL1 | WRITE_REG | 0x080;
.var powerdown_rx_buf0a[4]; // rx dma dummy buffer not used for anything;
// AD1836 codec register commands - Serial SPI 16-bit Word Format as follows:
// D15 to D12 = Codec Register Address
// D11 = Read/Write register (1=rd, 0=wr)
// D10 = reserved bit, clear to zero
// D9 to D0 = Data Field for codec register
.var tx_buf3a[21] = //program register commands
DAC_CONTROL1 | WRITE_REG | 0x000, // we "OR" in address, rd/wr, and register data
DAC_CONTROL1 | WRITE_REG | 0x000, // for ease in reading register values
DAC_CONTROL2 | WRITE_REG | 0x000, // write DAC_CTL1 twice to workaround pwdwn SPI anomaly
DAC_VOLUME0 | WRITE_REG | 0x3FF,
DAC_VOLUME1 | WRITE_REG | 0x3FF,
DAC_VOLUME2 | WRITE_REG | 0x3FF,
DAC_VOLUME3 | WRITE_REG | 0x3FF,
DAC_VOLUME4 | WRITE_REG | 0x3FF,
DAC_VOLUME5 | WRITE_REG | 0x3FF,
ADC_CONTROL1 | WRITE_REG | 0x000, // write ADC_CTL1 twice to workaround pwdwn SPI
anomaly
ADC_CONTROL1 | WRITE_REG | 0x000,
ADC_CONTROL3 | WRITE_REG | 0x000, // 256*Fs Clock Mode !!!, differential PGA mode
ADC_CONTROL2 | WRITE_REG | 0x380, // SOUT MODE = 110 --> TDM Mode, Master device
ADC_CONTROL2 | WRITE_REG | 0x380,
// read register commands
ADC0_PEAK_LEVEL | READ_REG | 0x000, // status will be in rx_buf1a[13-19] memory locations
ADC1_PEAK_LEVEL | READ_REG | 0x000,
ADC2_PEAK_LEVEL | READ_REG | 0x000,
ADC3_PEAK_LEVEL | READ_REG | 0x000,
ADC_CONTROL1 | READ_REG | 0x000,
ADC_CONTROL2 | READ_REG | 0x000,
ADC_CONTROL3 | READ_REG | 0x000;
.var rx_buf1a[21];
/* ISR counters, for debug purposes to see how many times SPORT DMA interrupts are serviced */
.VAR SP1I_counter = 0;
.VAR SP3I_counter = 0;
.section /pm pm_code;
////////////////////////////////////////////////////////////////////////////////
// //
// Program SPORT1 & SPORT3 Control Registers for SPI emulation control //
// //
////////////////////////////////////////////////////////////////////////////////
Program_AD1836_regs_via_SPI:
r0=0x00000000; // initially clear SPORT control register
dm(SPCTL1)=r0;
dm(SPCTL3)=r0;
ustat1=dm(SPCTL3);
ustat2=dm(SPCTL1);
ustat3=dm(SP13MCTL);
powerdown_reset_AD1836:
r0=powerdown_AD1836; dm(II3A)=r0; /* Internal DMA6 memory address */
r0=1; dm(IM3A)=r0; /* Internal DMA6 memory access modifier */
r0=@powerdown_AD1836; dm(C3A)=r0; /* Contains number of DMA6 transfers to be done */
r0=powerdown_rx_buf0a; dm(II1A)=r0; /* Internal DMA2 memory address */
r0=1; dm(IM1A)=r0; /* Internal DMA2 memory access modifier */
r0=@powerdown_rx_buf0a; dm(C1A)=r0; /* Contains number of DMA2 transfers to be done */
/* clear multichannel/miscellaneous control register for SPORT1 & SPORT3 */
R0 = 0x0; dm(SP13MCTL) = R0;
R0 = 0x0011002B; dm(DIV3) = R0;
R0 = 0; dm(DIV1) = R0;
bit set ustat1 DDIR | SDEN_A | LAFS | LFS | IFS | FSR | CKRE | ICLK | SLEN16 | SPEN_A;
dm(SPCTL3) = ustat1;
bit set ustat2 SDEN_A | LAFS | LFS | FSR | CKRE | SLEN16 | SPEN_A;
bit clr ustat2 DDIR | IFS | ICLK;
dm(SPCTL1) = ustat2;
bit set imask SP1I | SP3I; // enable SPORT1 RX and SPORT3 TX interrupts
powerdwm_not_done_yet:
idle;
R1 = 0x00000008; // Test for SPORT3
R0 = DM(DMASTAT);
R0 = R0 AND R1;
IF NE jump powerdwm_not_done_yet;
bit clr imask SP1I|SP3I; // disable SPORT1 RX and SPORT3 TX interrupts
Wait_Approx_1s:
lcntr = 3000, do waitloop until lce;
nop;
```

```
nop;
nop;
waitloop: nop;
bit clr ustat1 0xFFFFFFFF;
dm(SPCTL1) = ustat1;
dm(SPCTL3) = ustat1;
IRPTL=0;
bit clr IMASK SP1I | SP3I;
SPORT_DMA_setup:
r0=0x00000000; // initially clear SPORT control register
dm(SPCTL1)=r0;
dm(SPCTL3)=r0;
ustat1=dm(SPCTL3);
ustat2=dm(SPCTL1);
ustat3=dm(SP13MCTL);
r0=tx_buf3a; dm(II3A)=r0; /* Internal DMA6 memory address */
r0=1; dm(IM3A)=r0; /* Internal DMA6 memory access modifier */
r0=@tx_buf3a; dm(C3A)=r0; /* Contains number of DMA6 transfers to be done */
r0=rx_buf1a; dm(II1A)=r0; /* Internal DMA2 memory address */
r0=1; dm(IM1A)=r0; /* Internal DMA2 memory access modifier */
r0=@rx_buf1a; dm(C1A)=r0; /* Contains number of DMA2 transfers to be done */
/* clear multichannel/miscellaneous control register for SPORT1 & SPORT3 */
R0 = 0x0; dm(SP13MCTL) = R0;
/*internally generating FS3 and *HARDWARE* loop it back to FS1 for SPORT1/3 SPI control*/
/*
Maximum SPI serial bit clock rate is 8MHz... select 1 MHz to allow safety factor of 8
according to 9-42 of user's manual, xCLKDIV = ((2 x fCLKIN)/(serial clock frequency)) - 1
Thus, xCLKDIV = ((2 x 30 MHz)/(1 MHz)) - 1 = 59 = 0011 1011 (binary) = 0x003B
Now, since we want 16 bit clocks per frame, set FSDIV to 16-1 = 15 = 0x000F */
// R0 = 0x00270004; dm(DIV3) = R0;
// R0 = 0x000F003B; dm(DIV3) = R0;
R0 = 0x0011002B; dm(DIV3) = R0;
R0 = 0; dm(DIV1) = R0;
/* SPCTL3 SPORT CONTROL REGISTER BITs
31:26 -- Read-only status bits
25 -- DDIR Bit = 1, Transmitter
24 -- SPORT Enable B: 0 Disabled
23 -- reserved
22 -- Word Select: 0 issue if data in either Tx
21:20 -- DMA chaining and DMA enables for Channel B: 0:0 Disabled
19 -- SPORT xmit DMA chaining enable A: 0 disable
18 -- SPORT xmit DMA enable A: 0 disable
17 -- Late FS: 1 Late (see p.7 of 1836 data sheet)
16 -- Active Low FS: 1 Active Low
15 -- TFS data dependency: 0 TFS signal generated only when new data is in
SPORT channel's transmit data buffer
14 -- IFS Source: 1 internal
13 -- FSR Requirement: 1 FS required
12 -- Active Clock Edge: 1 rising edge
11 -- Operation mode: 0 non-I2S mode
10 -- Xmit Clk source: 1 internal
9 -- 16/32-bit pack: 0 no unpacking of 32-bit words into separate 16-bit
words for transmission
8:4 -- Serial Word Length minus 1: 01111
3 -- Endian word format: 0 MSB first
2:1 -- Data Type: 0:0 r-justify; fill MSBs w/0s
0 -- SPORT Enable A: 1 enable A */
// R0 = 0x020374F1;
bit set ustat1 DDIR | SDEN_A | LAFS | LFS | IFS | FSR | CKRE | ICLK | SLEN16 | SPEN_A;
dm(SPCTL3) = ustat1;
/* SRCTL0 SPORT CONTROL REGISTER BITs
31:26 -- Read-only status bits
25 -- DDIR Bit = 0, Receiver
24 -- SPORT Enable B: 0 Disabled
23 -- MCE - SPORT Mode: 0 DSP SPORT Mode
22 -- SPORT Loopback: 0 disable
21:20 -- DMA chaining and DMA enables for Channel B: 0:0 Disabled
19 -- SPORT Rcv DMA chaining enable A: 0 disabled
18 -- SPORT Rcv DMA enable A: 0 disabled
17 -- Late RFS: 1 Late (see p.7 of 1836 data sheet)
16 -- Active Low RFS: 1 Active Low (again, see p.7 of...)
15 -- reserved
14 -- IRFS - RFS Source: 0 external
13 -- RFS Requirement: 1 RFS required
12 -- Active Clock Edge: 1 rising edge
11 -- Operation mode: 0 non-I2S mode
10 -- Rcv Clk source: 0 external
```

```
9 -- 16/32-bit pack: 0 no packing of received 16-bit words into 32-bit words
8:4 -- Serial Word Length minus 1: 01111
3 -- Endian word format: 0 MSB first
2:1 -- Data Type: 0:0 r-justify; fill MSBs w/0s
0 -- SPORT Enable A: 1 enable A
NOTE: SPORT1 & SPORT3 clock and frame syncs tied together, generated by SPORT3 */
// R0 = 0x000330F1;
bit set ustat2 SDEN_A | LAFS | LFS | FSR | CKRE | SLEN16 | SPEN_A;
bit clr ustat2 DDIR | IFS | ICLK;
dm(SPCTL1) = ustat2;
bit set imask SP1I | SP3I; // enable SPORT0 RX and SPORT2 TX interrupts
SPORT_DMAs_not_done_yet:
idle;
R1 = 0x0000000A; // Test for SPORT1 and SPORT3 DMA completion
R0 = DM(DMASTAT);
R0 = R0 AND R1;
IF NE jump SPORT_DMAs_not_done_yet;
bit clr ustat1 0xFFFFFFFF;
dm(SPCTL1) = ustat1;
dm(SPCTL3) = ustat1;
IRPTL=0;
bit clr IMASK SP1I | SP3I; // disable SPORT1 and SPORT3 interrupts
rts;
/* With lcntr = 1000, this actually waits roughly 1s*/
Wait_Approx_1500ms:
lcntr = 1000, do waitloop250ms until lce;
nop; nop; nop;
//call Wait_Approx_999us;
nop; nop;
waitloop250ms: nop;
nop; nop;
rts;
/* This loop has been cut to approx length 110ms instead of 167ms*/
Wait_Approx_167ms:
lcntr = 110, do waitloop200ms until lce;
nop; nop; nop;
//call Wait_Approx_999us;
nop; nop;
waitloop200ms: nop;
nop; nop;
rts;
////////////////////////////////////////////////////////////
// //
// SPORT1 and SPORT3 Interrupt Service Routines //
// //
////////////////////////////////////////////////////////////
Count_SPORT1_RX_IRQs:
r0=dm(SP1I_counter); /* get last count */
r0=r0+1; /* increment count */
dm(SP1I_counter)=r0; /* save updated count */
RTI;
Count_SPORT3_TX_IRQs:
r0=dm(SP3I_counter); /* get last count */
r0=r0+1; /* increment count */
dm(SP3I_counter)=r0; /* save updated count */
RTI;
```

SPORTx IOP Register Clear Init Routine

```
/* ////////////////////////////////////////////////////////////////////////////// /
/ ROUTINE TO CLEAR AND RESET ALL SPORT1 REGISTERS /
```

```
/ /
/ /
/ This routine simply clears all SPORT0/1 ctrl and DMA registers back to their /
/ default states so that we can reconfigure it for our AD1836 application. /
/ /
/ John Tomarakos /
/ ADI DSP Applications /
/ Rev 1.0 /
/ 4/30/99 /
/ / */
/* ////////////////////////////////////////////////////////////////////////////////
*/
/* ADSP-21161 system Register bit definitions */
#include "def21161.h"
.GLOBAL Clear_All_SPT_Regs;
.section /pm pm_code;
Clear_All_SPT_Regs:
IRPTL = 0x00000000; /* clear pending interrupts */
bit clr imask SP0I|SP1I|SP2I|SP3I;
R0 = 0x00000000;
dm(SPCTL0) = R0; /* sport0 control register */
dm(SPCTL1) = R0; /* sport1 control register */
dm(SPCTL2) = R0; /* sport3 control register */
dm(SPCTL3) = R0; /* sport4 control register */
dm(DIV0) = R0; /* sport0 frame sync divide register */
dm(DIV1) = R0; /* sport1 frame sync divide register */
dm(DIV2) = R0; /* sport2 frame sync divide register */
dm(DIV3) = R0; /* sport3 frame sync divide register */
/* SPORT 0 & 2 Miscellaneous Control Bits Registers */
R0 = 0x00000000; /* Enable SPORT loopback bit 12 */
dm(SP02MCTL) = R0;
dm(SP13MCTL) = R0;
/* sport0 receive multichannel word enable registers */
R0 = 0x00000000; /* multichannel mode disabled */
dm(MR0CS0) = R0;
dm(MR0CS1) = R0;
dm(MR0CS2) = R0;
dm(MR0CS3) = R0;
/* sport1 receive multichannel word enable registers */
R0 = 0x00000000; /* multichannel mode disabled */
dm(MR1CS0) = R0;
dm(MR1CS1) = R0;
dm(MR1CS2) = R0;
dm(MR1CS3) = R0;
/* sport2 transmit multichannel word enable registers */
R0 = 0x00000000; /* multichannel mode disabled */
dm(MT2CS0) = R0;
dm(MT2CS1) = R0;
dm(MT2CS2) = R0;
dm(MT2CS3) = R0;
/* sport3 transmit multichannel word enable registers */
R0 = 0x00000000; /* multichannel mode disabled */
dm(MT3CS0) = R0;
dm(MT3CS1) = R0;
dm(MT3CS2) = R0;
dm(MT3CS3) = R0;
/* sport0 receive multichannel companding enable registers */
R0 = 0x00000000; /* no companding */
dm(MR0CCS0) = R0;
dm(MR0CCS1) = R0;
dm(MR0CCS2) = R0;
dm(MR0CCS3) = R0;
/* sport1 receive multichannel companding enable registers */
R0 = 0x00000000; /* no companding */
dm(MR1CCS0) = R0;
dm(MR1CCS1) = R0;
dm(MR1CCS2) = R0;
dm(MR1CCS3) = R0;
/* sport2 transmit multichannel companding enable registers */
R0 = 0x00000000; /* no companding */
dm(MT2CCS0) = R0;
dm(MT2CCS1) = R0;
dm(MT2CCS2) = R0;
dm(MT2CCS3) = R0;
/* sport3 transmit multichannel companding enable registers */
R0 = 0x00000000; /* no companding */
dm(MT3CCS0) = R0;
```

```
dm(MT3CCS1) = R0;
dm(MT3CCS2) = R0;
dm(MT3CCS3) = R0;
RTS;
```

SPORT0 TDM Receive Interrupt Service Routine

```
/*****************************************************************************************
*******
/ /
/ AD1836 - SPORT0 RX INTERRUPT SERVICE ROUTINE /
/ /
/ Receives input data from the 2 AD1836 ADCs via SPORT1 and transmits processed audio data /
/ back out to the 3 AD1836 Stereo DACs/Line Outputs /
/ /
*****************************************************************************************
*******
/ /
/ This Serial Port 0 Recieve Interrupt Service Routine performs arithmetic computations on /
/ the SPORT1 receive DMA buffer (rx_buf) and places results to SPORT1 transmit DMA buffer
(tx_buf)/
/ /
/ rx0a_buf[8] - DSP SPORT recieve buffer /
/ Slot # Description DSP Data Memory Address /
/ ------ ------------------------------------ ------------------------------------------
- /
/ 0 Internal ADC 0 Left Channel DM(rx0a_buf + 0) = DM(rx0a_buf + Internal_ADC_L0) /
/ 1 Internal ADC 1 Left Channel DM(rx0a_buf + 1) = DM(rx0a_buf + Internal_ADC_L1) /
/ 2 External Auxilliary ADC 0 Left Chan. DM(rx0a_buf + 2) = DM(rx0a_buf + AUX_ADC_L0) /
/ 3 External Auxilliary ADC 1 Left Chan. DM(rx0a_buf + 3) = DM(rx0a_buf + AUX_ADC_L1) /
/ 4 Internal ADC 1 Right Channel DM(rx0a_buf + 4) = DM(rx0a_buf + Internal_ADC_R0) /
/ 5 Internal ADC 1 Right Channel DM(rx0a_buf + 5) = DM(rx0a_buf + Internal_ADC_R1) /
/ 6 External Auxilliary ADC 0 Right Chan. DM(rx0a_buf + 6) = DM(rx0a_buf + AUX_DAC_R0) /
/ 7 External Auxilliary ADC 1 Right Chan. DM(rx0a_buf + 7) = DM(rx0a_buf + AUX_DAC_R1) /
/ /
/ tx2a_buf[8] - DSP SPORT transmit buffer /
/ Slot # Description DSP Data Memory Address /
/ ------ ------------------------------------ ------------------------------------------
- /
/ 0 Internal DAC 0 Left Channel DM(tx0a_buf + 0) = DM(tx0a_buf + Internal_DAC_L0) /
/ 1 Internal DAC 1 Left Channel DM(tx0a_buf + 1) = DM(tx0a_buf + Internal_DAC_L1) /
/ 2 Internal DAC 2 Left Channel DM(tx0a_buf + 2) = DM(tx0a_buf + Internal_DAC_L2) /
/ 3 External Auxilliary DAC 0 Left Chan. DM(rx0a_buf + 3) = DM(tx0a_buf + AUX_DAC_L0) /
/ 4 Internal DAC 0 Right Channel DM(tx0a_buf + 4) = DM(tx0a_buf + Internal_DAC_R0) /
/ 5 Internal DAC 1 Right Channel DM(tx0a_buf + 5) = DM(tx0a_buf + Internal_DAC_R1) /
/ 6 Internal DAC 2 Left Channel DM(tx0a_buf + 6) = DM(tx0a_buf + Internal_DAC_R3) /
/ 7 External Auxilliary DAC 0 Right Chan. DM(tx0a_buf + 7) = DM(tx0a_buf + AUX_DAC_R0) /
/ /
*****************************************************************************************
******/
/* ADSP-21161 System Register bit definitions */
#include "def21161.h"
/* AD1836 TDM Timeslot Definitions */
/* 8 successive 32 bit samples (representing 8 channels of audio data) make up the 256 bit TDM
frame */
#define Internal_ADC_L0 0
#define Internal_ADC_L1 1
#define AUX_ADC_L0 2
#define AUX_ADC_L1 3
#define Internal_ADC_R0 4
#define Internal_ADC_R1 5
#define AUX_ADC_R0 6
#define AUX_ADC_R1 7
#define Internal_DAC_L0 0
#define Internal_DAC_L1 1
#define Internal_DAC_L2 2
#define AUX_DAC_L0 3
#define Internal_DAC_R0 4
#define Internal_DAC_R1 5
#define Internal_DAC_R2 6
#define AUX_DAC_R0 7
.GLOBAL Process_AD1836_Audio_Samples; /* Label of code listed here to get samples into and out
of*/
/* SPORT DMA buffers and process_audio routine*/
.GLOBAL Left_Channel_In0; /* These will tranfer ADC samples of interest from the*/
```

```
.GLOBAL Right_Channel_In0; /* Process_AD1836_Audio_Samples routine to the process_audio
routine*/
.GLOBAL Left_Channel_In1;
.GLOBAL Right_Channel_In1;
.GLOBAL Left_Channel_SPDIF_rx;
.GLOBAL Right_Channel_SPDIF_rx;
.GLOBAL Left_Channel_Out0; /* These will bring samples for the DACs back into
the*/
.GLOBAL Right_Channel_Out0; /* Process_AD1836_Audio_Samples routine from the
process_audio routine*/
.GLOBAL Left_Channel_Out1;
.GLOBAL Right_Channel_Out1;
.GLOBAL Left_Channel_Out2;
.GLOBAL Right_Channel_Out2;
.GLOBAL Left_Channel_AD1852;
.GLOBAL Right_Channel_AD1852;
.GLOBAL Left_Channel_Out_Thru; /* These serve the same function for a different set of audio
*/
.GLOBAL Right_Channel_Out_Thru;/* samples destined for the DACs*/
//.EXTERN tx2a_buf; /* These are the DMA buffers that hold the 8 channels of audio */
//.EXTERN rx0a_buf; /* immediately pre-transmission and post-reception */
//.EXTERN process_audio; /* Label of audio algorithm code listed in another file which
executes */
/* a procesing alogorithm on the audio data before it is output */
/* to the DACs */
.section /dm dm_codec;
/* AD1836 stereo-channel data holders - used for DSP processing of audio data received from
codec */
.VAR Left_Channel_In0; /* Input values from AD1836 ADCs */
.VAR Left_Channel_In1;
.VAR Right_Channel_In0;
.VAR Right_Channel_In1;
.VAR Left_Channel_SPDIF_rx;
.VAR Right_Channel_SPDIF_rx;
.VAR Left_Channel_Out0; /* Output values for AD1836 DACs */
.VAR Left_Channel_Out1;
.VAR Left_Channel_Out2;
.VAR Right_Channel_Out0;
.VAR Right_Channel_Out1;
.VAR Right_Channel_Out2;
.VAR Left_Channel_AD1852;
.VAR Right_Channel_AD1852;
.VAR Left_Channel; /* can use for intermediate results to next filter stage */
.VAR Right_Channel; /* can use for intermediate results to next filter stage */
/* TDM audio frame/ISR counter, for debug purposes */
.VAR audio_frame_timer = 0;
.section /pm pm_code;
Process_AD1836_Audio_Samples:
/* get AD1836 left channel input samples, save to data holders for processing */
r0 = dm(rx0a_buf + Internal_ADC_L0); dm(Left_Channel_In0) = r0;
r0 = dm(rx0a_buf + Internal_ADC_L1); dm(Left_Channel_In1) = r0;
r0 = dm(rx0a_buf + AUX_ADC_L0); dm(Left_Channel_SPDIF_rx) = r0;
/* get AD1836 right channel input samples, save to data holders for processing */
r0 = dm(rx0a_buf + Internal_ADC_R0); dm(Right_Channel_In0) = r0;
r0 = dm(rx0a_buf + Internal_ADC_R1); dm(Right_Channel_In1) = r0;
r0 = dm(rx0a_buf + AUX_ADC_R0); dm(Right_Channel_SPDIF_rx) = r0;
do_audio_processing:
call (pc, process_audio);
/* ---- DSP processing is finished, now playback results to AD1836 ---- */
playback_AD1836_left_DACs: /* output processed left ch audio samples to AD1836 */
r0 = dm(Left_Channel_Out0); dm(tx2a_buf + Internal_DAC_L0) = r0;
r1 = dm(Left_Channel_Out1); dm(tx2a_buf + Internal_DAC_L1) = r1;
r2 = dm(Left_Channel_Out2); dm(tx2a_buf + Internal_DAC_L2) = r2;
r3 = dm(Left_Channel_AD1852); dm(tx2a_buf + AUX_DAC_L0) = r3;
playback_AD1836_right_DACs: /* output processed right ch audio samples to AD1836 */
r0 = dm(Right_Channel_Out0); dm(tx2a_buf + Internal_DAC_R0) = r0;
r1 = dm(Right_Channel_Out1); dm(tx2a_buf + Internal_DAC_R1) = r1;
r2 = dm(Right_Channel_Out2); dm(tx2a_buf + Internal_DAC_R2) = r2;
r3 = dm(Right_Channel_AD1852); dm(tx2a_buf + AUX_DAC_R0) = r3;
tx_done:
r0=dm(audio_frame_timer); /* get last count */
rti(db); /* return from interrupt, delayed branch */
r0=r0+1; /* increment count */
dm(audio_frame_timer)=r0; /* save updated count */
/* ////////////////////////////////////////////////////////////////////////////// */
```

APPENDIX B:
C Program Source Code Listing for 21161 EZ-KIT Lite Audio Driver (Visual DSP Project Files)

Main.C

```
#include "ADDS_21161_EzKit.h"
#include <def21161.h>
#include <signal.h>
float * DelayLine;
int Index = 0;
void Process_Samples( int sig_int)
{
Receive_Samples();
/* Perform AD1836/AD1852/SPDIF Audio Processing Here */
// left channel 1/8th inch jack to headphone out left channel
Left_Channel_Out0 = Left_Channel_In1;
// left channel 1/8th inch jack to headphone out left channel
Right_Channel_Out0 = Right_Channel_In1;
/* create a simple stereo digital delay on internal AD1836 stereo DAC1 channel */
Right_Channel_Out1 = DelayLine[Index] + Right_Channel_In1; // delayed left + right channel
Left_Channel_Out1 = Left_Channel_In1; // loopback left channel, no processing
DelayLine[Index++] = Left_Channel_In1; // store left channel into delay-line
if (Index == 12000) Index = 0;
/* loop back other audio data */
Left_Channel_Out2 = Left_Channel_In0;
Right_Channel_Out2 = Right_Channel_In0;
Left_Channel_AD1852 = Left_Channel_SPDIF_rx;
Right_Channel_AD1852 = Right_Channel_SPDIF_rx;
Transmit_Samples();
}
void main()
{
/* Setup Interrupt edges and flag I/O directions */
Setup_ADSP21161N();
/* Setup SDRAM Controller */
Setup_SDRAM();
Setup_AD1836();
Init_AD1852_DACs();
Program_SPORT02_TDM_Registers();
Program_SPORT02_DMA_Channels();
interruptf( SIG_SP0I, Process_Samples);
*(int *) SP02MCTL |= MCE;
DelayLine = (float *) 0x00200000;
for (;;)
asm("idle;");
```

C-callable Assembly Routines for Processing Codec Data

```
#include <asm_sprt.h>
#include <def21161.h>
#include "adds_21161_ezkit.h"
.segment /dm seg_dmda;
/* AD1836 stereo-channel data holders - used for DSP processing of audio data received from
codec */
// input channels
.var _Left_Channel_In0; /* Input values from the 2 AD1836 internal stereo ADCs */
.var _Left_Channel_In1; /* 1/8th inch stereo jack connected to internal stereo ADC1
*/
.var _Right_Channel_In0;
.var _Right_Channel_In1;
.var _Left_Channel_SPDIF_rx;/* Input values from the DAR CS8414 */
.var _Right_Channel_SPDIF_rx;
//output channels
```

```
.var _Left_Channel_Out0; /* Output values for the 3 AD1836 internal stereo DACs */
.var _Left_Channel_Out1; /* Left and Right Channel 0 DACs go to headphone jack */
.var _Left_Channel_Out2;
.var _Right_Channel_Out0;
.var _Right_Channel_Out1;
.var _Right_Channel_Out2;
.var _Left_Channel_AD1852; /* Output values for AD1852 stereo DAC */
.var _Right_Channel_AD1852;
.var _Left_Channel; /* Can use these variables as intermediate results to next
filtering stage */
.var _Right_Channel;
.global _Left_Channel_In0;
.global _Left_Channel_In1;
.global _Right_Channel_In0;
.global _Right_Channel_In1;
.global _Left_Channel_Out0;
.global _Left_Channel_Out1;
.global _Left_Channel_Out2;
.global _Right_Channel_Out0;
.global _Right_Channel_Out1;
.global _Right_Channel_Out2;
.global _Left_Channel_AD1852;
.global _Right_Channel_AD1852;
.global _Left_Channel_SPDIF_rx;
.global _Right_Channel_SPDIF_rx;
.extern _rx0a_buf;
.extern _tx2a_buf;
.endseg;
.segment /pm seg_pmco;
_Receive_Samples:
.global _Receive_Samples;
//void Receive_Samples();
/* get AD1836 left channel input samples, save to data holders for processing */
r1 = -31;
r0 = dm(_rx0a_buf + Internal_ADC_L0); f0 = float r0 by r1; dm(_Left_Channel_In0) = r0;
r0 = dm(_rx0a_buf + Internal_ADC_L1); f0 = float r0 by r1; dm(_Left_Channel_In1) = r0;
r0 = dm(_rx0a_buf + AUX_ADC_L0); f0 = float r0 by r1; dm(_Left_Channel_SPDIF_rx) = r0;
/* get AD1836 right channel input samples, save to data holders for processing */
r0 = dm(_rx0a_buf + Internal_ADC_R0); f0 = float r0 by r1; dm(_Right_Channel_In0) = r0;
r0 = dm(_rx0a_buf + Internal_ADC_R1); f0 = float r0 by r1; dm(_Right_Channel_In1) = r0;
r0 = dm(_rx0a_buf + AUX_ADC_R0); f0 = float r0 by r1; dm(_Right_Channel_SPDIF_rx) = r0;
leaf_exit;
_Transmit_Samples:
.global _Transmit_Samples;
r1 = 31;
/* output processed left ch audio samples to AD1836 */
r0 = dm(_Left_Channel_Out0); r0 = trunc f0 by r1; dm(_tx2a_buf + Internal_DAC_L0) = r0;
r0 = dm(_Left_Channel_Out1); r0 = trunc f0 by r1; dm(_tx2a_buf + Internal_DAC_L1) = r0;
r0 = dm(_Left_Channel_Out2); r0 = trunc f0 by r1; dm(_tx2a_buf + Internal_DAC_L2) = r0;
r0 = dm(_Left_Channel_AD1852); r0 = trunc f0 by r1; dm(_tx2a_buf + AUX_DAC_L0) = r0;
/* output processed right ch audio samples to AD1836 */
r0 = dm(_Right_Channel_Out0); r0 = trunc f0 by r1; dm(_tx2a_buf + Internal_DAC_R0) = r0;
r0 = dm(_Right_Channel_Out1); r0 = trunc f0 by r1; dm(_tx2a_buf + Internal_DAC_R1) = r0;
r0 = dm(_Right_Channel_Out2); r0 = trunc f0 by r1; dm(_tx2a_buf + Internal_DAC_R2) = r0;
r0 = dm(_Right_Channel_AD1852); r0 = trunc f0 by r1; dm(_tx2a_buf + AUX_DAC_R0) = r0;
leaf_exit;
.endseg;
```

C code for DSP System & Codec Initialization Routines

```
#include "ADDS_21161_EzKit.h"
#include <def21161.h>
#include <signal.h>
/*****************************************************************************************
*******
/ /
/ AD1836 - SETUP and Data Routing /
/ /
/ Receives input data from the 2 AD1836 ADCs via SPORT1 and transmits processed audio data /
/ back out to the 3 AD1836 Stereo DACs/Line Outputs /
/ /
*****************************************************************************************
*******
/ /
```

```
/ This Serial Port 0 Recieve Interrupt Service Routine performs arithmetic computations on /
/ the SPORT1 receive DMA buffer (rx_buf) and places results to SPORT1 transmit DMA buffer
(tx_buf)/
/ /
/ rx0a_buf[8] - DSP SPORT recieve buffer /
/ Slot # Description DSP Data Memory Address /
/ ------ ------------------------------------ ------------------------------------------
- /
/ 0 Internal ADC 0 Left Channel DM(rx0a_buf + 0) = DM(rx0a_buf + Internal_ADC_L0) /
/ 1 Internal ADC 1 Left Channel DM(rx0a_buf + 1) = DM(rx0a_buf + Internal_ADC_L1) /
/ 2 External Auxilliary ADC 0 Left Chan. DM(rx0a_buf + 2) = DM(rx0a_buf + AUX_ADC_L0) /
/ 3 External Auxilliary ADC 1 Left Chan. DM(rx0a_buf + 3) = DM(rx0a_buf + AUX_ADC_L1) /
/ 4 Internal ADC 1 Right Channel DM(rx0a_buf + 4) = DM(rx0a_buf + Internal_ADC_R0) /
/ 5 Internal ADC 1 Right Channel DM(rx0a_buf + 5) = DM(rx0a_buf + Internal_ADC_R1) /
/ 6 External Auxilliary ADC 0 Right Chan. DM(rx0a_buf + 6) = DM(rx0a_buf + AUX_DAC_R0) /
/ 7 External Auxilliary ADC 1 Right Chan. DM(rx0a_buf + 7) = DM(rx0a_buf + AUX_DAC_R1) /
/ /
/ tx2a_buf[8] - DSP SPORT transmit buffer /
/ Slot # Description DSP Data Memory Address /
/ ------ ------------------------------------ ------------------------------------------
- /
/ 0 Internal DAC 0 Left Channel DM(tx0a_buf + 0) = DM(tx0a_buf + Internal_DAC_L0) /
/ 1 Internal DAC 1 Left Channel DM(tx0a_buf + 1) = DM(tx0a_buf + Internal_DAC_L1) /
/ 2 Internal DAC 2 Left Channel DM(tx0a_buf + 2) = DM(tx0a_buf + Internal_DAC_L2) /
/ 3 External Auxilliary DAC 0 Left Chan. DM(rx0a_buf + 3) = DM(tx0a_buf + AUX_DAC_L0) /
/ 4 Internal DAC 0 Right Channel DM(tx0a_buf + 4) = DM(tx0a_buf + Internal_DAC_R0) /
/ 5 Internal DAC 1 Right Channel DM(tx0a_buf + 5) = DM(tx0a_buf + Internal_DAC_R1) /
/ 6 Internal DAC 2 Left Channel DM(tx0a_buf + 6) = DM(tx0a_buf + Internal_DAC_R3) /
/ 7 External Auxilliary DAC 0 Right Chan. DM(tx0a_buf + 7) = DM(tx0a_buf + AUX_DAC_R0) /
/ /
**********************************************************************************************
******/
int powerdown_AD1836[4] = {DAC_CONTROL1 | WRITE_REG | 0x004,
DAC_CONTROL1 | WRITE_REG | 0x004,
ADC_CONTROL1 | WRITE_REG | 0x080,
ADC_CONTROL1 | WRITE_REG | 0x080};
int powerdown_rx_buf0a[4]; // rx dma dummy buffer not used for anything;
// AD1836 codec register commands - Serial SPI 16-bit Word Format as follows:
// D15 to D12 = Codec Register Address
// D11 = Read/Write register (1=rd, 0=wr
// D10 = reserved bit, clear to zero
// D9 to D0 = Data Field for codec register
#define TX_BUF3A_LEN 21
int tx_buf3a[TX_BUF3A_LEN] = //program register commands
{DAC_CONTROL1 | WRITE_REG | 0x000, // we "OR" in address, rd/wr, and register data
DAC_CONTROL1 | WRITE_REG | 0x000, // for ease in reading register values
DAC_CONTROL2 | WRITE_REG | 0x000, // write DAC_CTL1 twice to workaround pwdwn SPI anomaly
DAC_VOLUME0 | WRITE_REG | 0x3FF,
DAC_VOLUME1 | WRITE_REG | 0x3FF,
DAC_VOLUME2 | WRITE_REG | 0x3FF,
DAC_VOLUME3 | WRITE_REG | 0x3FF,
DAC_VOLUME4 | WRITE_REG | 0x3FF,
DAC_VOLUME5 | WRITE_REG | 0x3FF,
ADC_CONTROL1 | WRITE_REG | 0x000, // write ADC_CTL1 twice to workaround pwdwn SPI anomaly
ADC_CONTROL1 | WRITE_REG | 0x000,
ADC_CONTROL3 | WRITE_REG | 0x000, // 256*Fs Clock Mode !!!, differential PGA mode
ADC_CONTROL2 | WRITE_REG | 0x380, // SOUT MODE = 110 --> TDM Mode, Master device
ADC_CONTROL2 | WRITE_REG | 0x380,
// read register commands
ADC0_PEAK_LEVEL | READ_REG | 0x000, // status will be in rx_buf1a[13-19] memory locations
ADC1_PEAK_LEVEL | READ_REG | 0x000,
ADC2_PEAK_LEVEL | READ_REG | 0x000,
ADC3_PEAK_LEVEL | READ_REG | 0x000,
ADC_CONTROL1 | READ_REG | 0x000,
ADC_CONTROL2 | READ_REG | 0x000,
ADC_CONTROL3 | READ_REG | 0x000 };
#define RX_BUF1A_LEN 21
int rx_buf1a[RX_BUF1A_LEN];
void SPORT_RX_IRQ( int sig_int)
{}
int Setup_AD1836()
{
int i;
/* Powerdown reset of AD1836 */
*(int *) II3A = (int) powerdown_AD1836;
*(int *) IM3A = 1;
*(int *) C3A = 4;
```

```
*(int *) II1A = (int) powerdown_rx_buf0a;
*(int *) IM1A = 1;
*(int *) C1A = 4;
*(int *) SP13MCTL = 0;
*(int *) DIV3 = 0x0011002B;
*(int *) DIV1 = 0;
*(int *) SPCTL3 |= DDIR | SDEN_A | LAFS | LFS | IFS | FSR | CKRE | ICLK | SLEN16 | SPEN_A;
*(int *) SPCTL1 |= SDEN_A | LAFS | LFS | FSR | CKRE | SLEN16 | SPEN_A;
*(int *) SPCTL1 &= (~DDIR & ~IFS & ~ICLK);
interruptf( SIG_SP1I, SPORT_RX_IRQ);
interruptf( SIG_SP3I, SPORT_RX_IRQ);
while ( (*(int*) DMASTAT) & 0x8 )
asm("idle;");
interruptf( SIG_SP1I, SIG_IGN);
interruptf( SIG_SP3I, SIG_IGN);
/* Now, stall for about 1 second */
for (i=0;i<3000;i++)
asm("nop; nop; nop; nop; nop;");
*(int *) SPCTL1 = 0;
*(int *) SPCTL3 = 0;
/* SPORT DMA Setup */
*(int *) II3A = (int) tx_buf3a;
*(int *) IM3A = 1;
*(int *) C3A = TX_BUF3A_LEN;
*(int *) II1A = (int) rx_buf1a;
*(int *) IM1A = 1;
*(int *) C1A = RX_BUF1A_LEN;
*(int *) SP13MCTL = 0;
*(int *) DIV3 = 0x0011002B;
*(int *) DIV1 = 0;
*(int *) SPCTL3 |= DDIR | SDEN_A | LAFS | LFS | IFS | FSR | CKRE | ICLK | SLEN16 | SPEN_A;
*(int *) SPCTL1 |= SDEN_A | LAFS | LFS | FSR | CKRE | SLEN16 | SPEN_A;
*(int *) SPCTL1 &= (~DDIR & ~IFS & ~ICLK);
interruptf( SIG_SP1I, SPORT_RX_IRQ);
interruptf( SIG_SP3I, SPORT_RX_IRQ);
while ( (*(int*) DMASTAT) & 0xA )
asm("idle;");
interruptf( SIG_SP1I, SIG_IGN);
interruptf( SIG_SP3I, SIG_IGN);
*(int *) SPCTL1 = 0;
*(int *) SPCTL3 = 0;
return 0;
}
int rx0a_buf[8]; /* receive buffer (DMA)*/
int tx2a_buf[8] = { 0x00000000, /* transmit buffer (DMA)*/
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x00000000,
0x00000000};
/* TCB = "Transfer Control Block" */
/* TCB format: ECx (length of destination buffer),
EMx (destination buffer step size),
EIx (destination buffer index (initialized to start address)),
GPx ("general purpose"),
CPx ("Chain Point register"; points to last address (IIx) of
next TCB to jump to
upon completion of this TCB.),
Cx (length of source buffer),
IMx (source buffer step size),
IIx (source buffer index (initialized to start address)) */
int rcv0a_tcb[8] = {0, 0, 0, 0, 0, 8, 1, (int) rx0a_buf}; /* SPORT0 receive tcb */
int xmit2a_tcb[8] = {0, 0, 0, 0, 0, 8, 1, (int) tx2a_buf}; /* SPORT2 transmit tcb */
void Program_SPORT02_TDM_Registers()
{
*(int *) DIV0 = 0;
*(int *) DIV2 = 0;
/* SPORT0 and SPORT2 are being operated in "multichannel" mode.
This is synonymous with TDM mode which is the operating mode for the AD1836 */
/* SPORT 0&2 Miscellaneous Control Bits Registers */
/* SP02MCTL = 0x000000E2, Hold off on MCM enable, and number of TDM slots to 8 active channels
*/
/* Multichannel Frame Delay=1, Number of Channels = 8, LB disabled */
*(int *) SP02MCTL = NCH_8 | MFD1;
/* sport0 control register set up as a receiver in MCM */
```

```
/* sport 0 control register SPCTL0 = 0x000C01F0 */
*(int *) SPCTL0 = SCHEN_A | SDEN_A | SLEN32;
/* sport2 control register set up as a transmitter in MCM */
/* sport 2 control register, SPCTL2 = 0x000C01F0 */
*(int *) SPCTL2 = SCHEN_A | SDEN_A | SLEN32;
/* sport0 & sport2 receive and transmit multichannel word enable registers */
/* enable receive channels 0-7 */
/* enable transmit channels 0-7 */
*(int *) MR0CS0 = *(int *) MT2CS0 =0x000000FF;
/* sport0 & sport2 receive & transmit multichannel companding enable registers */
/* no companding for our 8 active timeslots*/
/* no companding on SPORT0 receive */
/* no companding on SPORT2 transmit */
*(int *) MR0CCS0 = *(int *) MT2CCS0 = 0;
}
void Program_SPORT02_DMA_Channels()
{
xmit2a_tcb[4] = *(int *) CP2A = ((int) xmit2a_tcb + 7) & 0x3FFFF | (1<<18);
rcv0a_tcb[4] = *(int *) CP0A = ((int) rcv0a_tcb + 7) & 0x3FFFF | (1<<18);
}
#ifdef DEBUG
/* TDM audio frame/ISR counter, for debug purposes */
int audio_frame_timer = 0;
#endif
/* AD1852 Setup */
#define SPI_TX_BUF_LEN 5
int spi_tx_buf[SPI_TX_BUF_LEN] = { RESET_AD1852 | CONTROL_REG, // reset AD1852
DEASSERT_RESET | CONTROL_REG, // remove reset command
WL_24_BIT_DATA | I2S_JUSTIFIED | NO_DEMPH_FILTER | CONTROL_REG,
0x00FC | VOLUME_LEFT,
0x00FC | VOLUME_RIGHT };
void Init_AD1852_DACs()
{
// initially clear SPI control register
*(int *) SPICTL = 0;
*(int *) IISTX = (int) spi_tx_buf;
*(int *) IMSTX = 1;
*(int *) CSTX = SPI_TX_BUF_LEN;
asm("#include <def21161_may2001.h>");
asm("bit set LIRPTL SPITMSK;");
interruptf( SIG_LP0I, SPORT_RX_IRQ);
*(int *) SPICTL |= SPIEN|SPTINT|TDMAEN|MS|FLS1|CPHASE|DF|WL16|BAUDR4|PSSE|DCPH0|SGN|GM;
*(int *) SPICTL &= (~CP & ~FLS0 & ~FLS2 & ~FLS3 & ~SMLS & ~DMISO & ~OPD & ~PACKEN & ~SENDZ &
~RDMAEN & ~SPRINT);
while( (*(int*) DMASTAT) & DMA9ST) idle();
interruptf( SIG_LP0I, SIG_IGN);
asm("bit clr LIRPTL SPITMSK;");
*(int *) SPICTL = 0;
}
/* SDRAM Setup Routine */
void Setup_SDRAM()
{
/*clear MSx waitstate and mode*/
*(int *)WAIT &= 0xFFF00000;
/*refresh rate*/
*(int*) SDRDIV = 0x1000;
// SDCTL = 0x02014231;
// 1/2 CCLK, no SDRAM buffering option, 2 SDRAM banks
// SDRAM mapped to bank 0 only, no self-refresh, page size 256 words
// SDRAM powerup mode is prechrg, 8 CRB refs, and then mode reg set cmd
// tRCD = 2 cycles, tRP=2 cycles, tRAS=3 cycles, SDCL=1 cycle
// SDCLK0, SDCLK1, RAS, CAS and SDCLKE activated
*(int *) SDCTL |= SDTRCD2|SDCKR_DIV2|SDBN2|SDEM0|SDPSS|SDPGS256|SDTRP2|SDTRAS3|SDCL1;
*(int *) SDCTL &= ~SDBUF & ~SDEM3 & ~SDEM2 & ~SDEM1 & ~SDSRF & ~SDPM & ~DSDCK1 & ~DSDCTL;
}
/* DSP Setup */
void Setup_ADSP21161N()
{
/* *** Enable circular buffering in MODE1 Register for revision 0.x silicon.
Important when porting 2106x code!!! */
asm("bit set MODE1 CBUFEN;");
/* Setup hardware interrupts, FLAG LEDs and pushbutton */
*(int *) IOFLAG = FLG9|FLG8|FLG7|FLG6|FLG5|FLG4|FLG9O|FLG8O|FLG7O|FLG6O|FLG5O|FLG4O;
/* flag 0-3 are inputs from pushbutton switches */
asm("bit clr MODE2 FLG0O | FLG1O | FLG2O | FLG3O;");
/* irqx edge sensitive */
asm("bit set mode2 IRQ2E | IRQ0E | IRQ1E;");
```

```
}
void Blink_LED_Test( int interations )
{
int i,k;
for(i=0;i<interations;i++)
{
*(int*) IOFLAG ^= FLG9|FLG8|FLG7|FLG6|FLG5|FLG4;
for (k=0;k<10000000;k++) {}
}
}
```

C code for DSP System & Codec Initialization Routines

```
#ifdef __ECC__
/* Insert C Definitions here.... */
int Setup_AD1836();
void Program_SPORT02_TDM_Registers();
void Program_SPORT02_DMA_Channels();
void Receive_Samples();
void Transmit_Samples();
void Init_AD1852_DACs();
void Setup_SDRAM();
void Setup_ADSP21161N();
void Blink_LED_Test( int interations );
extern float Left_Channel0; /* Input values from AD1836 ADCs */
extern float Left_Channel1;
extern float Left_Channel2;
extern float Left_Channel3; /* AD1852 */
extern float Right_Channel0;
extern float Right_Channel1;
extern float Right_Channel2;
extern float Right_Channel3; /* AD1852 */
extern float Left_Channel_SPDIF_rx;
extern float Right_Channel_SPDIF_rx;
// input channels
extern float Left_Channel_In0; /* Input values from the 2 AD1836 internal stereo ADCs */
extern float Left_Channel_In1; /* 1/8th inch stereo jack connected to internal stereo ADC1 */
extern float Right_Channel_In0;
extern float Right_Channel_In1;
extern float Left_Channel_SPDIF_rx; /* Input values from the DAR CS8414 */
extern float Right_Channel_SPDIF_rx;
//output channels
extern float Left_Channel_Out0; /* Output values for the 3 AD1836 internal stereo DACs */
extern float Left_Channel_Out1; /* Left and Right Channel 0 DACs go to headphone jack */
extern float Left_Channel_Out2;
extern float Right_Channel_Out0;
extern float Right_Channel_Out1;
extern float Right_Channel_Out2;
extern float Left_Channel_AD1852; /* Output values for AD1852 stereo DAC */
extern float Right_Channel_AD1852;
#else
/* Insert Assembly Definitions here.... */
#endif
/* Insert global definitions here */
// AD1836 codec SPI control/status register definitions
#define READ_REG 0x0800
#define WRITE_REG 0x0000
#define DAC_CONTROL1 0x0000
#define DAC_CONTROL2 0x1000
#define DAC_VOLUME0 0x2000
#define DAC_VOLUME1 0x3000
#define DAC_VOLUME2 0x4000
#define DAC_VOLUME3 0x5000
#define DAC_VOLUME4 0x6000
#define DAC_VOLUME5 0x7000
#define ADC0_PEAK_LEVEL 0x8000
#define ADC1_PEAK_LEVEL 0x9000
#define ADC2_PEAK_LEVEL 0xA000
#define ADC3_PEAK_LEVEL 0xB000
#define ADC_CONTROL1 0xC000
#define ADC_CONTROL2 0xD000
#define ADC_CONTROL3 0xE000
#define RESERVED_REG 0xF000
#define NCH_8 0x000000E0 /* Number of MCM channels - 1 */
#define Initialize_TDM 0xD380 /* 1101 0001 1000 0000*/
/* AD1836 TDM Timeslot Definitions */
#define Internal_ADC_L0 0
#define Internal_ADC_L1 1
```

```
#define AUX_ADC_L0 2
#define AUX_ADC_L1 3
#define Internal_ADC_R0 4
#define Internal_ADC_R1 5
#define AUX_ADC_R0 6
#define AUX_ADC_R1 7
#define Internal_DAC_L0 0
#define Internal_DAC_L1 1
#define Internal_DAC_L2 2
#define AUX_DAC_L0 3
#define Internal_DAC_R0 4
#define Internal_DAC_R1 5
#define Internal_DAC_R2 6
#define AUX_DAC_R0 7
/* Leave a safety margin of 5x for the 1836 */
#define AD1836_RESET_CYCLES 300
#define AD1836_WARMUP_CYCLES 60000
/* AD1852 Defines */
#define VOLUME_LEFT 0x0
#define VOLUME_RIGHT 0x2
#define CONTROL_REG 0x1
// AD1852 control word parameters
#define DEASSERT_RESET 0x0000
#define INTERP2xMODE 0x0800
#define INTERP4xMODE 0x0400
#define WL_24_BIT_DATA 0x0000
#define WL_20_BIT_DATA 0x0100
#define WL_16_BIT_DATA 0x0200
#define RESET_AD1852 0x0080
#define SOFT_MUTE 0x0040
#define RIGHT_JUSTIFIED 0x0000
#define I2S_JUSTIFIED 0x0010
#define LEFT_JUSTIFIED 0x0020
#define DSP_SERIAL 0x0030
#define NO_DEMPH_FILTER 0x0000
#define FILTER_44_1_kHz 0x0004
#define FILTER_32_kHz 0x0008
#define FILTER_48_kHz 0x000C
/* SDRAM Defines */
#define sdram_size 0xfffff
```

ADSP-21161 Interrupt Vector Table

```
/* ********************************************************************************** */
/* */
/* ADSP-21161 INTERRUPT VECTOR TABLE */
/* */
/* For use with the 21161 EZ-KIT Lite */
/* */
/* ADI DSP Central Applications Engineering */
/* 10/3/00 */
/* ********************************************************************************** */
.EXTERN _main;
.EXTERN Init_DSP;
.EXTERN Count_SPORT1_RX_IRQs;
.EXTERN Count_SPORT3_TX_IRQs;
.EXTERN Process_AD1836_Audio_Samples;
.section /dm dm_data;
/* SPORT TX ISR counter, for debug purposes */
.VAR SPORT2_TXDMA_counter = 0;
.section/PM isr_tbl; /* 21161 Interrupt Service Table */
/* 0x00 Reserved Interrupt */
/* 0x00 0x01 0x02 0x03 */
/* NOP; NOP; NOP; NOP; */ // Reserved interrupt
// Vector for RESET:
/* 0x04 - reset vector starts at location 0x40005 */
RSTI_scv: /* IDLE; */ // Implicit IDLE instruction for boot kernel cleanup
call Init_DSP;
NOP;
jump _main;
/* 0x08 - Vector address for illegal input condition detected */
IICD_svc: RTI; RTI; RTI; RTI;
/* 0x0C - Vector address for status stack/loop stack overflow or PC stack full: */
SOVFI_svc: RTI; RTI; RTI; RTI;
```

```
// 0x10 - Vector address for high priority timer interrupt:
TMZHI_svc: RTI; RTI; RTI; RTI;
// 0x14 - Vector address for Vector Interrupt:
VIRPTI_svc: RTI; RTI; RTI; RTI;
// 0x18 - Vector address for Hardware Interrupt 2 (IRQ2):
IRQ2I_svc: RTI; RTI; RTI; RTI;;
// 0x1C - Vector address for Hardware Interrupt 1 (IRQ1):
IRQ1I_svc: RTI; RTI; RTI; RTI;
// 0x20 - Vector address for Hardware Interrupt 0 (IRQ0):
IRQ0I_svc: RTI; RTI; RTI; RTI;
/* 0x24 - Reserved interrupt */
reserved_0x24: NOP; NOP; NOP; NOP; // Reserved interrupt
// Vectors for Serial port DMA channels:
/* 0x28 - Vector address for serial port 0 primary A, secondary B RX/TX buffers (DMA Channels
0 & 1) */
SP0I_svc: JUMP Process_AD1836_Audio_Samples;
RTI; RTI; RTI;
/* 0x2C - Vector address for serial port 1 primary A, secondary B RX/TX buffers (DMA Channel 2
& 3) */
SP1I_svc: JUMP Count_SPORT1_RX_IRQs;
RTI; RTI; RTI;
/* 0x30 - Vector address for serial port 2 primary A, secondary B RX/TX buffers (DMA Channel 4
& 5) */
SP2I_svc: r0=dm(SPORT2_TXDMA_counter); /* get last count */
RTI(db);
r0=r0+1; /* increment count */
dm(SPORT2_TXDMA_counter)=r0; /* save updated count */
/* 0x34 - Vector address for serial port 3 primary A, secondary B RX/TX buffers (DMA Channel 6
& 7) */
SP3I_svc: JUMP Count_SPORT3_TX_IRQs;
RTI; RTI; RTI;
// Vectors for link port DMA channels:
/* 0x38 - Vector address for Link Buffer 0 (DMA Channel 8) */
LP0I_svc: RTI; RTI; RTI; RTI;
/* 0x3C - Vector address for Link Buffer 1 (DMA Channel 9) */
LP1I_svc: RTI; RTI; RTI; RTI;
/* 0x40 - Vector address for SPI Receive (DMA Channel 8) */
SPIRI_svc: RTI; RTI; RTI; RTI;
/* 0x44 - Vector address for SPI Receive (DMA Channel 9) */
SPITI_svc: RTI; RTI; RTI; RTI;
/* 0x48 - Reserved Interrupt */
reserved_0x48: RTI; RTI; RTI; RTI;
/* 0x4C - Reserved Interrupt */
reserved_0x4C: RTI; RTI; RTI; RTI;
// Vectors for External port DMA channels:
/* 0x50 - Vector address for External Port Buffer 0 (DMA Channel 10) */
EP0I_svc: RTI; RTI; RTI; RTI;
/* 0x54 - Vector address for External Port Buffer 0 (DMA Channel 11) */
EP1I_svc: RTI; RTI; RTI; RTI;
/* 0x58 - Vector address for External Port Buffer 0 (DMA Channel 12) */
EP2I_svc: RTI; RTI; RTI; RTI;
/* 0x5C - Vector address for External Port Buffer 0 (DMA Channel 13) */
EP3I_svc: RTI; RTI; RTI; RTI;
// 0x60 - Vector address for Link service request:
LSRQI_svc: RTI; RTI; RTI; RTI;
// 0x64 - Vector address for DAG1 buffer 7 circular buffer overflow:
CB7I_svc: RTI; RTI; RTI; RTI;
// 0x68 - Vector address for DAG2 buffer 15 circular buffer overflow:
CB15I_svc: RTI; RTI; RTI; RTI;
// 0x6C - Vector address for lower priority timer interrupt:
TMZLI_svc: RTI; RTI; RTI; RTI;
// 0x70 - Vector address for fixed-point overflow interrupt:
FIXI_svc: RTI; RTI; RTI; RTI;
// 0x74 - Vector address for floating-point overflow exception interrupt:
FLTOI_svc: RTI; RTI; RTI; RTI;
// 0x78 - Vector address for floating-point underflow exception interrupt:
FLTUI_svc: RTI; RTI; RTI; RTI;
// 0x7C - Vector address for floating-point invalid exception interrupt:
FLTII_svc: RTI; RTI; RTI; RTI;
// 0x80 - Vector address for user software interrupt 0:
SFT0I_svc: RTI; RTI; RTI; RTI;
// 0x84 - Vector address for user software interrupt 1:
SFT1I_svc: RTI; RTI; RTI; RTI;
// 0x88 - Vector address for user software interrupt 2:
SFT2I_svc: RTI; RTI; RTI; RTI;
// 0x8C - Vector address for user software interrupt 3:
SFT3I_svc: RTI; RTI; RTI; RTI;
```

```
/* 0x90 - Reserved Interrupt */
reserved_0x90: RTI; RTI; RTI; RTI;
```

Visual DSP Tools (21161 EZ-KIT Lite) Linker Description File

```
ARCHITECTURE(ADSP-21161)
//
// ADSP-21161 Memory Map:
// ------------------------------------------------
// Internal memory 0x0000 0000 to 0x000f ffff
// ------------------------------------------------
// 0x0000 0000 to 0x0001 ffff IOP Regs
// Block 0 0x0002 0000 to 0x0002 1fff Long Word (64) Addresses
// 0x0002 2000 to 0x0002 7fff (reserved)
// Block 1 0x0002 8000 to 0x0002 9fff Long Word (64) Addresses
// 0x0002 a000 to 0x0003 ffff (reserved)
// Block 0 0x0004 0000 to 0x0004 3fff Normal Word (32/48) Addresses
// 0x0004 4000 to 0x0004 ffff (reserved)
// Block 1 0x0005 0000 to 0x0005 3fff Normal Word (32/48) Addresses
// 0x0005 4000 to 0x0007 ffff (reserved)
// Block 0 0x0008 0000 to 0x0008 7fff Short Word (16) Addresses
// 0x0008 8000 to 0x0009 ffff (reserved)
// Block 1 0x000a 0000 to 0x000a 7fff Short Word (16) Addresses
// 0x000a 8000 to 0x000f ffff (reserved)
// ------------------------------------------------
// Multiproc memory 0x0010 0000 to 0x007f ffff
// ------------------------------------------------
// 0x0010 0000 to 0x0011 ffff Hammerhead ID=001 Internal memory
// 0x0012 0000 to 0x0013 ffff Hammerhead ID=010 Internal memory
// 0x0014 0000 to 0x0015 ffff Hammerhead ID=011 Internal memory
// 0x0016 0000 to 0x0017 ffff Hammerhead ID=100 Internal memory
// 0x0018 0000 to 0x0019 ffff Hammerhead ID=101 Internal memory
// 0x001a 0000 to 0x001b ffff Hammerhead ID=110 Internal memory
// 0x001c 0000 to 0x001f ffff Hammerhead ID=all Internal memory
// ------------------------------------------------
// External memory 0x0020 0000 to 0xffff ffff
// ------------------------------------------------
//
// This architecture file allocates:
// Internal 256 words of run-time header in memory block 0
// 256 words of initialization code in memory block 0
// 6K words of C code space in memory block 0
// 1.5K words of C PM data space in memory block 0
// 8K words of C DM data space in memory block 1
// 4K words of C heap space in memory block 1
// 4K words of C stack space in memory block 1
SEARCH_DIR( $ADI_DSP\211xx\lib )
$LIBRARIES = libc160.dlb, libio160_32.dlb, libdsp160.dlb, libcpp.dlb, libcpprt.dlb;
// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
#ifdef __cplusplus
$OBJECTS = 160_cpp_hdr.doj, $COMMAND_LINE_OBJECTS;
#else
//$OBJECTS = 160_hdr.doj, $COMMAND_LINE_OBJECTS;
$OBJECTS = $COMMAND_LINE_OBJECTS;
#endif
MEMORY
{
seg_rth { TYPE(PM RAM) START(0x00040005) END(0x000400ff) WIDTH(48) }
seg_init { TYPE(PM RAM) START(0x00040100) END(0x000401ff) WIDTH(48) }
seg_pmco { TYPE(PM RAM) START(0x00040200) END(0x000419ff) WIDTH(48) }
seg_pmda { TYPE(PM RAM) START(0x00042a00) END(0x00043fff) WIDTH(32) }
#ifdef __cplusplus
seg_ctdm { TYPE(DM RAM) START(0x00050000) END(0x000500ff) WIDTH(32) }
seg_ctdmend { TYPE(DM RAM) START(0x00050100) END(0x000501ff) WIDTH(32) }
seg_dmda { TYPE(DM RAM) START(0x00050200) END(0x00051fff) WIDTH(32) }
#else
seg_dm48 { TYPE(PM RAM) START(0x00050000) END(0x000500ff) WIDTH(48) }
seg_dm32 { TYPE(DM RAM) START(0x00050200) END(0x000502ff) WIDTH(32) }
```

```
seg_dmda { TYPE(DM RAM) START(0x00050300) END(0x00051fff) WIDTH(32) }
#endif
seg_heap { TYPE(DM RAM) START(0x00052000) END(0x00052fff) WIDTH(32) }
seg_stak { TYPE(DM RAM) START(0x00053000) END(0x00053fff) WIDTH(32) }
// External Memory SDRAM Mapped to Bank 0
segsdram { TYPE(DM RAM) START(0x00200000) END(0x002FFFFF) WIDTH(32) }
}
PROCESSOR p0
{
LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
SECTIONS
{
// .text output section
seg_rth
{
INPUT_SECTIONS( $OBJECTS(isr_tbl) $LIBRARIES(isr_tbl))
} >seg_rth
seg_init
{
ldf_seginit_space = . ;
INPUT_SECTIONS( $OBJECTS(seg_init) $LIBRARIES(seg_init))
} >seg_init
seg_pmco
{
INPUT_SECTIONS( $OBJECTS(pm_code) $LIBRARIES(pm_code))
} >seg_pmco
seg_pmda
{
INPUT_SECTIONS( $OBJECTS(pm_data) $LIBRARIES(pm_data))
} >seg_pmda
#ifdef __cplusplus
seg_ctdm
{
__ctors = .; /* points to the start of the section */
INPUT_SECTIONS( $OBJECTS(seg_ctdm) $LIBRARIES(seg_ctdm))
} > seg_ctdm
seg_ctdmend
{
INPUT_SECTIONS( $OBJECTS(seg_ctdmend) $LIBRARIES(seg_ctdmend))
} > seg_ctdmend
#endif
seg_dm48
{
INPUT_SECTIONS( $OBJECTS(seg_dm48) $LIBRARIES(seg_dm48))
} > seg_dm48
seg_dm32
{
INPUT_SECTIONS( $OBJECTS(seg_dm32) $LIBRARIES(seg_dm32))
} > seg_dm32
seg_dmda
{
INPUT_SECTIONS( $OBJECTS(dm_data dm_codec) $LIBRARIES(dm_data))
} > seg_dmda
stackseg
{
// allocate a stack for the application
ldf_stack_space = .;
ldf_stack_length = MEMORY_SIZEOF(seg_stak);
} > seg_stak
heap
{
// allocate a heap for the application
ldf_heap_space = .;
ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(seg_heap) - 1;
ldf_heap_length = ldf_heap_end - ldf_heap_space;
} > seg_heap
segsdram SHT_NOBITS
{
INPUT_SECTIONS( $OBJECTS(segsdram) $LIBRARIES(segsdram))
} > segsdram
}
}
```