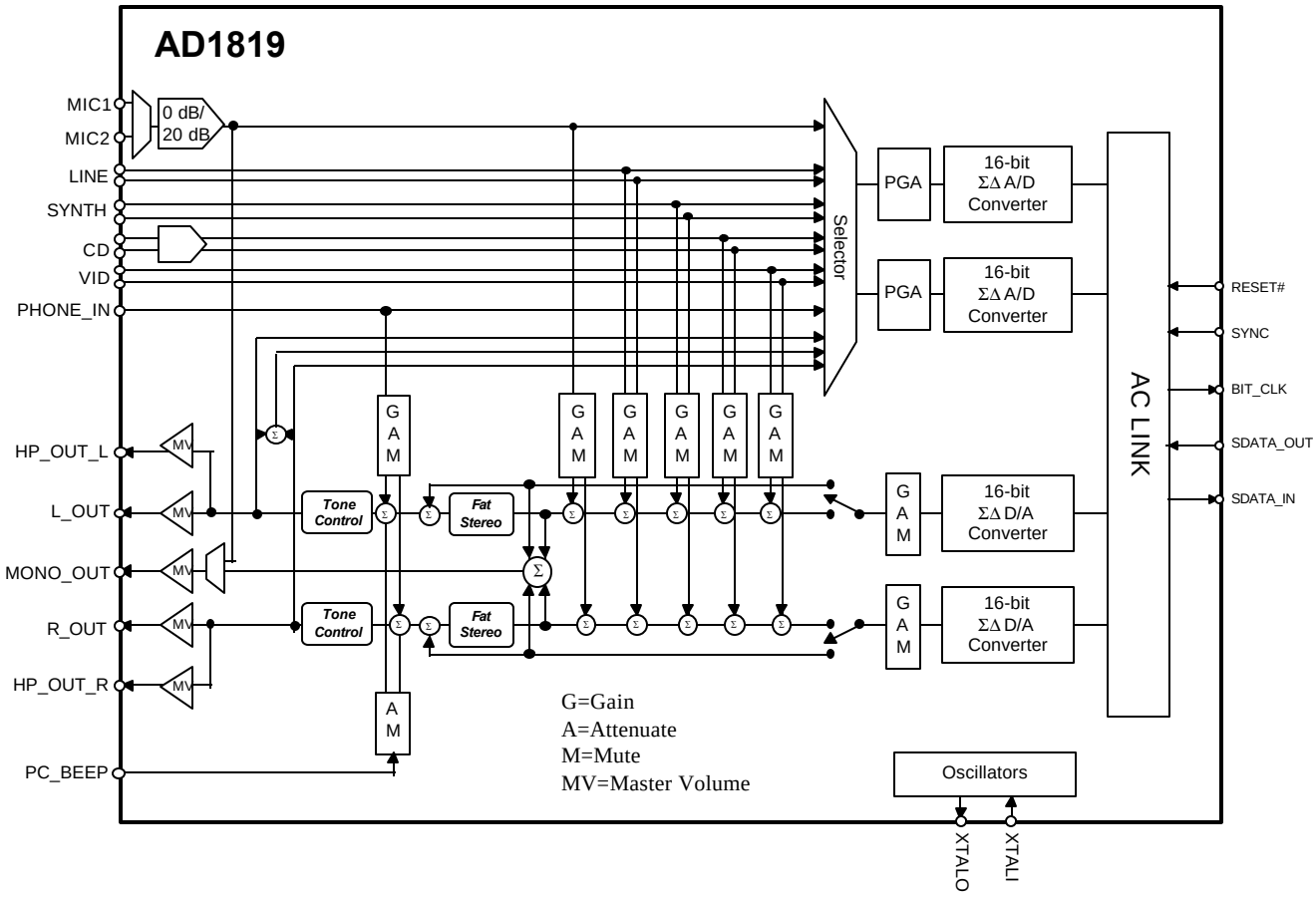
Interfacing the ADSP-21065L SHARC DSP to the AD1819A 'AC-97' SoundPort Codec

Codec interface driver recommendations for use with the ADSP-21065L EZ-LAB 's AD1819A … as well as other Analog Devices AC'97-compatible codecs such as the AD1819, AD1819B, AD1881, AD1881A, AD1882 and AD1885

Version 2.4A

John Tomarakos
ADI DSP Applications
10/12/99



0. Introduction

The AD1819x (AD1819, AD1819A and AD1819B) SoundPort Codec is a fully compliant AC'97 Analog Front End that can be used for processing or playback of analog signals in personal computers. Audio Codec '97 codecs are used on PC soundcards and PC motherboard's to interface with an AC-97 digital controller/accelerator, or directly to the PC motherboard's 64-bit microprocessor for 'native signal processing' support. With extended codec features that have been added to the baseline AC'97 specification, the AD1819x can also be easily interfaced to an Analog Devices ADSP-21xx and ADSP-21xxx DSP, thus providing additional flexibility of using the codec in an embedded low-cost audio application. The AD1819x can also provide a V.34 compatible modem analog front-end by supporting modem sample rates and filtering.

This application note will describe the how to interface the low-cost 32-bit SHARC DSP, the ADSP-21065L, to up to three daisy chained AD1819As per SPORT for use in an audio system. The DSP programming recommendations are also applicable to code compatible SHARC processors such as the ADSP-21061/61L, ADSP-21062/62L ADSP-21060/60L and the ADSP-21160. Using multiple AD1819s gives the DSP system designer more flexibility for capture and playback of 'CD-Quality' audio by providing 6 inputs channel and 6 output channels per SPORT for processing of multiple audio signals concurrently. No additional hardware glue-logic is required for daisy-chaining, and the programming of multiple AD1819s is easily accomplished through a simple register command scheme. Single and multiple AD1819x driver source code examples are provided in the application note for reference purposes. This source code was tested using the 21065L EZ-LAB Development Platform, which includes an AD1819A as the analog interface. Triple-codec source was tested and verified using the SHARC EZ-LAB with the Triple AD1819 MAFE board. Single codec drivers can serve as a basis for a dual or triple AD1819x codec interface requiring very little modification of the driver to enable a multi-codec interface, due to the ability to broadcast similar commands to all AD1819s through shared codec register address & register data timeslots. ***The latest revision of the AD1819 is the AD1819B. All references to the AD1819 or AD1819x in this document refer to the AD1819A and AD1819B as well. The programming recommendations in this document are also applicable to the AD1881, AD1881A, AD1882 and AD1885 (AC-97 rev 2.1 compliant codecs).***

0.1 What The AD1819x Offers Above The Baseline AC-97 1.03 Specification

The AD1819 exceeds all AC'97 Version 1.03 Specifications and offers additional features that are useful for DSP interfacing. Some of these include:

- ***Slot-16 DSP Serial Mode for DSP Serial Port Multichannel Mode (TDM) Compatibility.***
  This mode ensures that all serial port time-slots are 16 bits, allowing a much easier interface to 16-bit/32-bit DSPs that support a TDM (time division multiplexed) interface. Slot-16 mode is useful since the TAG is always 16-bits and equal length 16-bit slots eases to use of serial port 'autobuffering' of data, or 'DMA chaining', along with the SPORT's Multichannel Mode TDM operation.

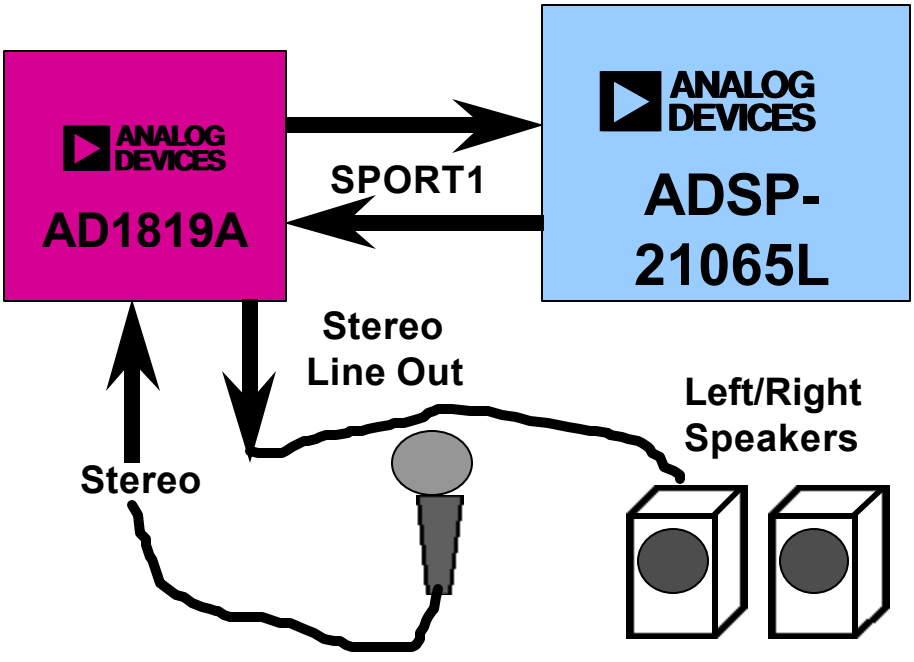
- ***Variable Sampling Rate Support On Both The Stereo Sigma-Delta ADCs and DACs***
  Variable sample rate allows you to 'record' and 'play back' audio signals at any sample rate from **7KHz to 48Khz in 1Hhz increments** with the use of two sample rate generator registers. The AD1819A can record and transmit ADC samples at one rate and play back received DAC samples at another rate. The left ADC and DAC channels can also be programmed to run at different rates as the right ADC and DAC channels. In addition to the 1Hz resolution the AD1819A also has a method for running at irrational modem rates by use of the 8/7 and 10/7 bits. To go with these modem sample rates the AD1819A has modem filters on the left channel. Please refer to our EE-Note #54 titled "*How To Use AD1819A Variable Sample Rate Support"* for additional information (located at Analog's web site: www.analog.com).

- ***High Quality AC-97 Output greater than 90 dB SNR***
  AC'97 Rev 1.03 defines at least 85 dB signal quality. The AD1819 exceeds this specification, providing greater than 90 dB dynamic range to provide near 'CD-Quality' sound with the use of Multibit Sigma Delta converter technology.

- ***Simple, Glueless Interface for Daisy-Chaining up to Three AD1819s***
  Three AD1819s can easily be interfaced to an Analog Devices DSP to provide 6 input channels and 6 output channels per SPORT. From a hardware standpoint, no additional glue circuitry is required for connection of multiple codecs. Each codec has 4 pins that are used for daisy-chaining up to 3 AD1819s: CS0, CS1, CHAIN_IN and CHAIN_CLK. From a software point of view, the DSP can communicate to all AD1819s at once, or can read/write codec registers to any desired codec at any time with the use of Mask Bits in the AD1819's Serial Configuration Register. Analog Devices was the only 1st generation AC-97 1.03 vendor to implement a simple multi-codec scheme.

- ***Phat Stereo 3D Enhancement***
  Provides a wider three dimensional sound in a stereo output field by giving the impression of spaciousness. The phase expansion capability allows the user to simulate the effect of the sound source coming from another direction other than the left and right stereo speaker sources.

21065L EZ-LAB / AD1819A Audio Development System



1. AD1819x / ADSP-2106x SHARC DSP Serial Interface Overview

The AD1819x (AD1819, AD1819A, AD1819B) Serial Port functionality is very similar other Analog Devices SoundPort Codecs like the AD1843 and AD1847. It's interface can communicates with an AC'97 controller, DSP or ASIC in a time-division multiplexed (TDM) mode, where codec register command/status information DAC/ADC data are received and transmitted in different timeslots in a TDM frame. The AD1819 communicates with the AC'97 controller via a digital serial link, which is referred to as the ***"AC-link."***

The AD1819x incorporates a 5 pin digital serial interface that links it to the AC'97 controller. The pins assigned to the AD1819x are: **SDATA_IN**, **SDATA_OUT**, **SYNC**, **BIT_CLK** and **RESET#**. All digital audio data and command/status information is communicated over this point to point serial interconnection to the DSP. A breakout of the signals connecting the two is shown in Figure 1. For a detailed description of the AC-link serial protocol between the DSP and the codec, the reader can refer to the next section.

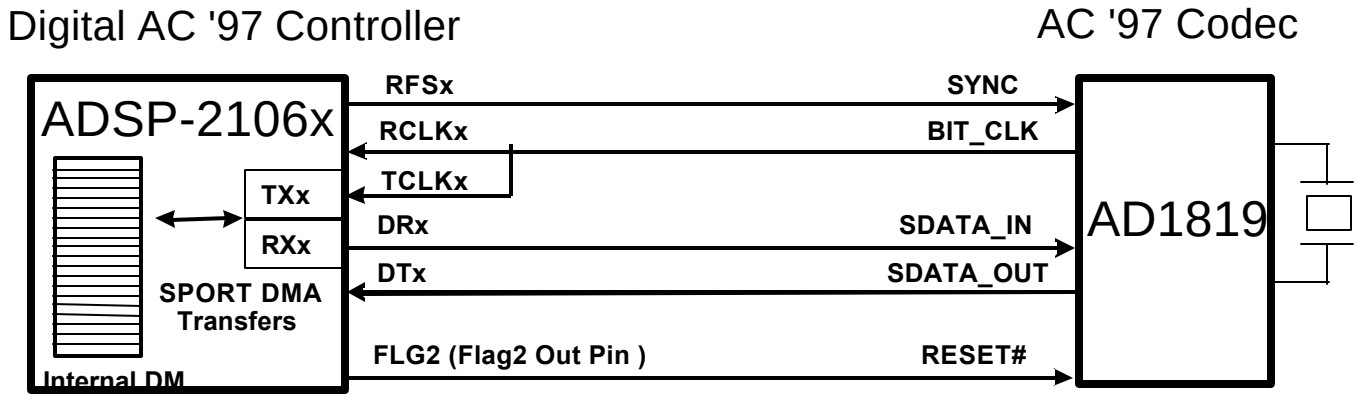


Figure 1. Example AD1819x Interconnection To The ADSP-2106x SHARC DSP's SPORT(0 or 1)

The AC-97 component specification defines digital serial connection known as an ***AC-Link***, which is a bi-directional, fixed rate, serial PCM digital stream. An AC'97-compatible codec handles multiple input and output audio streams, as well as command register accesses employing a time-division-multiplexed (TDM) scheme. The baseline AC-link architecture divides each audio frame into 12 outgoing and 12 incoming data streams, each with 20-bit sample resolution. The AD1819 also includes an additional mode of operation, considered to be an ***Enhanced AC-link Protocol Extension***, also referred to as ***SLOT-16 Mode***. This extension is very similar such that it also is also is a bi-directional, fixed rate, serial PCM digital stream. This Modified AC-link divides each audio frame into 16 outgoing and incoming data streams, each 16-bits per slot. This allows low DSP software overhead to transmit and receive data and thus enables a more simplified interface to an ADSP-21xx or ADSP-21xxx DSP. To achieve this, the AD1819's SLOT-16 Mode of Operation will place all DAC/ADC and command/status Timeslots to 16-bits to allow proper 16-bit TDM alignment for the DSP's serial port.

The AC-97 protocol could also be implemented with 18-bit or 20-bit DAC/ADC resolution with larger data word processors, given the headroom that the AC-link architecture provides. This application note will only assume 16-bit data by placing the AD1819x in SLOT-16 mode. As of this time, there is no performance benefit for using 18-bit or 20-bit words, since the use of the larger 20-bit timeslots will not necessarily improve the dynamic range and SNR. Also, the tag phase is always 16-bits, so all other larger word slots would be skewed relative to the DSP timeslot alignment. This would then require the DSP programmer to use shift/extract/deposit instructions on all data coming after the 1st 16-bit slot so that proper memory and register alignment occurs for all timeslot data. It is still possible to write a DSP driver that assumes 20-bit slots, although the DSP programmer would have to use additional instructions to ensure that data is packed and sent out properly. A 16-bit DSP cannot easily handle the additional headroom for 20-bit words, while a 32-bit DSP would have the overhead of packing and unpacking 20-bit data after the initial 16-bit timeslot. Again, since there is no SNR benefit of using larger data word timeslot sizes, the use of the AD1819 while not in SLOT-16 mode is not recommended for interfacing to a DSP TDM serial port as found in ADI's ADSP-21xx and ADSP-2106x DSPs.

1.1 AD1819x (AD1819/A/B) "AC-Link" Serial Port Clocks And Frame Sync Rates

To keep clock jitter to a minimum, the AD1819x derives its clock internally from an externally attached 24.576 MHz crystal (as required by the AC-97 specification), and drives a buffered and divided down (1/2) clock to the ADSP-2106x over AC-link under the signal name **BIT_CLK**. The crystal frequency can be different, but it would no longer be AC-97 compliant since it also affects actual value of the selected sample rate. Meeting AC-97 compliance is not necessary for embedded DSP designs (for tips on using a different crystal frequency, refer to EE-Note #53 up on the Analog Devices Web Site: www.analog.com). Clock jitter at the AD1819x DACs and ADCs is a fundamental impediment to high quality output, and the internally generated clock provided the AD1819x with a clean clock that is independent of the physical proximity of the ADSP-2106x processor. BIT_CLK, fixed at 12.288 MHz, provides the necessary clocking granularity to support 16, 16-bit outgoing and incoming time slots (12, 20-bit outgoing and incoming time slots in normal AC-97 mode). AC-link serial data is transitioned on each rising edge of BIT_CLK. The receiver of AC-link data, AD1819x for outgoing data and the ADSP-2106x for incoming data, samples each serial bit on the falling edges of BIT_CLK. The AD1819x drives the serial bit clock at 12.288 MHz, which the ADSP-2106x then qualifies with a synchronization signal to construct audio frames.

The beginning of all audio sample packets, or "***Audio Frames***", transferred over the AC-Link is synchronized to the rising edge of the **SYNC** signal. The SYNC pin is used for the serial interface frame synchronization and must be generated by the ADSP-2106x AC-97 Controller. Synchronization of all *AC-link* data transactions is signaled by the ADSP-2106x via the RFSx signal. SYNC, fixed at 48 kHz, is derived by dividing down the serial bit clock (BIT_CLK). The ADSP-2106x SHARC takes BIT_CLK (or RCLKx/TCLKx in SHARC DSP equivalent terms) as an input and generates SYNC (RFSx) by dividing BIT_CLK by 256. This yields a 48kHz SYNC signal whose period defines an audio frame, which is required to meet the AC-97 audio frame rate requirement. The SYNC (RFSx) pulse is driven by the ADSP-2106x processor by programming the RFSDIV register in the DSP. To generate a 48 kHz frame sync with an externally generated 12.288 MHz SCLK, the DSP must set a value of 255 (0x00FF) in the RFSDIV control register.

The **SDATA_IN** and **SDATA_OUT** pins handle the serial data input and output of the AD1819x. Both the AD1819x's SDATA_IN and SDATA_OUT pins transmit or receive data on 12 different timeslots (in addition to the Tag Phase) per frame in normal AC-97 mode, 16 different timeslots (1 Tag + 15 Data slots) in ***SLOT-16*** mode. The AD1819x transmits data on every rising edge of BIT_CLK (RCLKx/TCLKx) and it samples received data on the falling edge of BIT_CLK (RCLKx/TCLKx).

When the 48 kHz audio frame rate is not equivalent to the selected sample rate, then Valid Data Slot bits in the Tag Phase timeslot as well as the DAC request bits in the AD1819's Serial Configuration Register are used to control the sample data flow between the codec and the DSP. When the 48 kHz frame rate is equivalent to the converter sample rate, valid and request bits can be ignored since they will always be 1s.

2. AD1819x/ADSP2106x “AC-Link” Digital Serial Interface Protocol

The AC-link protocol described by the AC’97 specification provides for a special 16-bit time slot (**Slot 0**, often called the **'TAG Phase'**) wherein each bit conveys a valid tag for its corresponding time slot within the current audio frame. A “1” in a given bit position of slot 0 indicates that the corresponding time slot within the current audio frame has been assigned to a data stream, and contains valid data. If a slot is “tagged” invalid, it is the responsibility of the source of the data, (AD1819x for the input stream, ADSP-2106x for the output stream), to stuff all bit positions with 0’s during that slot’s active time. In the source code example in Appendix A, the ADSP-2106x processor ensures than invalid slots are stuffed with 0's.

SYNC can remain high for a total duration of 16 BIT_CLKs at the beginning of each audio frame, although for DSP interfacing, the ADI DSP usually pulses a frame sync for approximately 1 BIT_CLK, which is also acceptable for the AD1819x. The first timeslot portion of the audio frame is defined as the “**Tag Phase**”. The remainder of the audio frame is defined as the **"Data Phase."**

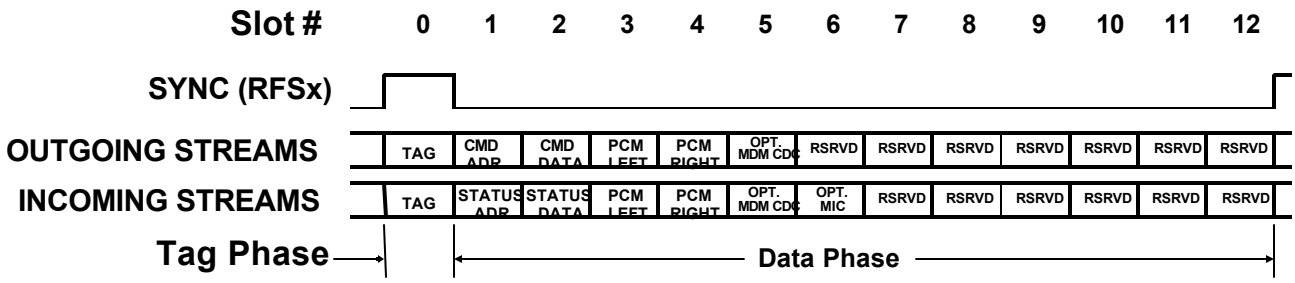


Figure 2. Standard AC'97 Version 1.03 Bi-directional Audio Frame

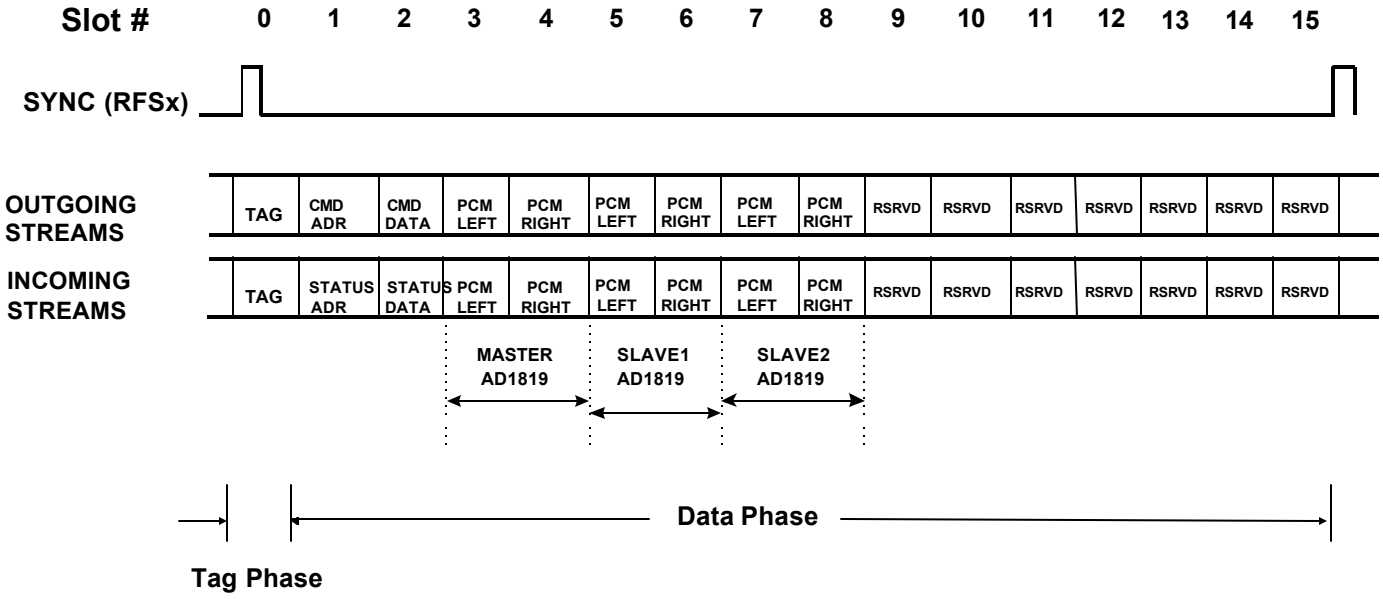


Figure 3. Modified AD1819x 'AC-97' Bi-directional Audio Frame Configured In SLOT-16 Mode

2.1 ADSP2106x / AD1819x Audio Output Frame (DTx to SDATA_OUT)

The audio output frame data streams correspond to the multiplexed bundles of all digital output data targeting the AD1819x DAC inputs, and control registers. Each audio output frame can support up to 16, 16-bit outgoing data time slots (by default, it is actually 12, 20-bit outgoing timeslots after the 16-bit slot 0 - the DSP must put the AD1819x into Slot-16 mode for SPORT compatibility). Slot 0 is a special reserved time slot containing 16 bits, which are used for AC-link protocol infrastructure.

Within slot 0 the first bit is a global bit (SDATA_OUT slot 0, bit 15) which flags the validity for the entire audio frame. If the **"Valid Frame"** bit is a 1, this indicates that the current audio frame contains at least one slot time of valid data. The next 12 bit positions sampled by the AD1819 indicate which of the corresponding 12 time slots contain valid data. Note that in Slot-16 mode, bit positions 13, 14 and 15 are always assumed to be zero. In this way data streams of differing sample rates can be transmitted across AC-link at its fixed 48 kHz audio frame rate. The timing diagram in Figure 4 illustrates the time slot based AC-link protocol in Slot-16 Mode. Serial timings in Figure 4 can assume either the SHARC's SPORT0 or SPORT1 pins. For SPORT0 we use RFS0 as the frame sync, TCLK0 as the serial clock, and DT0/DT0A as the data transmit pin. These timings also apply to SPORT1's RFS1 and DT1/DT1A pins, while the serial clock would correspond to TCLK1.

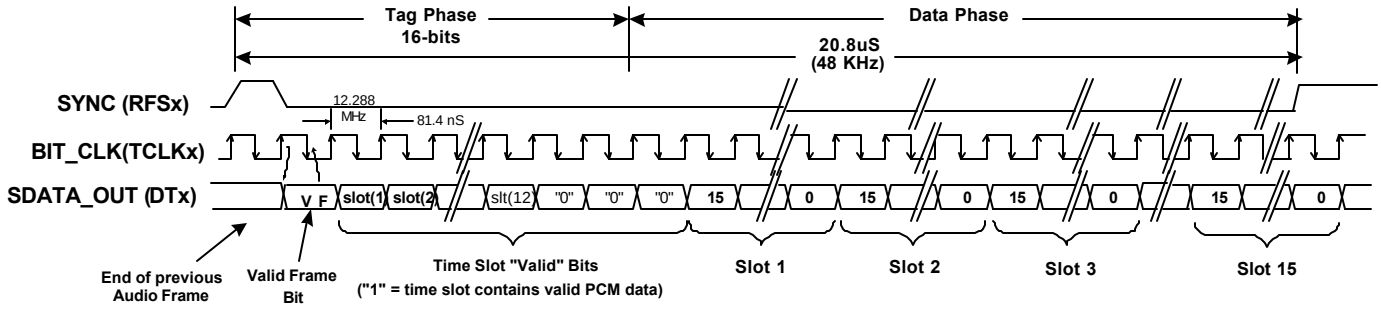


Figure 4. AD1819x Audio Output Frame in SLOT-16 Mode - ADSP-2106x to AD1819x Data Path **

**Note: The timing in figure 4 differs from the standard AC-97 timing of 12 slots, 20 bits in length for the data phase portion of the audio frame. The timeslots are set to 16 bits in length (SLOT16 Mode) by the ADSP-2106x during initial enabling of the DSP SPORT so that proper data alignment of TDM slots to occur. Also, the frame sync generated by the DSP is not set for the duration of the Tag Phase, as described in the AC-97 spec and AD1819 data sheet. The DSP generates a frame sync for approximately 1 serial clock cycle. However this does not affect the codec operation, since the codec samples the frame sync only for the first cycle prior to transmission of the MSB of the Tag Phase timeslot. Setup and hold times on the AD1819 are relaxed enough to meet the SHARC RFSx external generation timings listed in the ADSP-2106x data sheet.

A new audio output frame, shown in Figure 5, begins with a low to high transition of SYNC. SYNC is synchronous to the rising edge of BIT_CLK. On the immediately following falling edge of BIT_CLK, the AD1819x samples the assertion of SYNC. This falling edge marks the time when both sides of AC-link are aware of the start of a new audio frame. On the next rising of BIT_CLK, the ADSP-2106x transitions SDATA_OUT into the first bit position of slot 0 (**Valid Frame bit**). Each new bit position is presented to AC-link on a rising edge of BIT_CLK, and subsequently sampled by AD1819x on the following falling edge of BIT_CLK. This sequence ensures that data transitions and subsequent sample points for both incoming and outgoing data streams are time aligned.

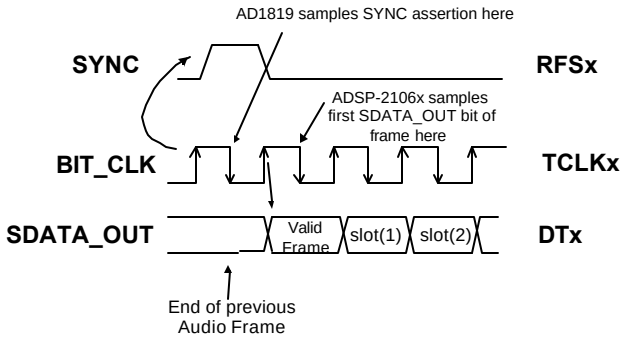


Figure 5. Start of an Audio Output Frame

SDATA_OUT's composite stream is MSB justified (MSB first) with all non-valid slots' bit positions stuffed with 0's by the ADSP-2106x. The DSP software can initialize the transmit DMA buffer to 0x0000s in the SPORT ISR. (Note that this is done in the SPORT1 transmit and received interrupt service routines shown in Appendix A)

In the event that there are less than 16 valid bits within an assigned and valid time slot, the ADSP-2106x should always stuff all trailing non-valid bit positions of the 16-bit slot with 0's.

When mono audio sample streams are output from the ADSP-2106x, it is necessary that BOTH left and right sample stream time slots be filled with the same data.

2.2 AD1819x/ADSP2106x Audio Input Frame (SDATA_IN to DRx)

The audio input frame data streams correspond to the multiplexed bundles of all digital input data targeting the ADSP-2106x. As is the case for audio output frame, each AD1819x audio input frame consists of 12, 16-bit time slots after the DSP programs the codec in SLOT-16 Mode. Slot 0 is a special reserved time slot containing 16 bits which are used for AC-link protocol infrastructure. The timing diagram in Figure 6 illustrates the time slot based AC-link protocol in Slot-16 Mode, and the Tag Phase's bit positions 13, 14 and 15 are zero. Serial timings in Figure 6 can assume either the SHARC's SPORT0 or SPORT1 pins. For SPORT0 we use RFS0 as the frame sync, RCLK0 as the serial clock and DR0/DR0A as the data receive pin. These timings also apply to SPORT1's RFS1 and DR1/DR1A pins, while the serial clock would correspond to RCLK1.

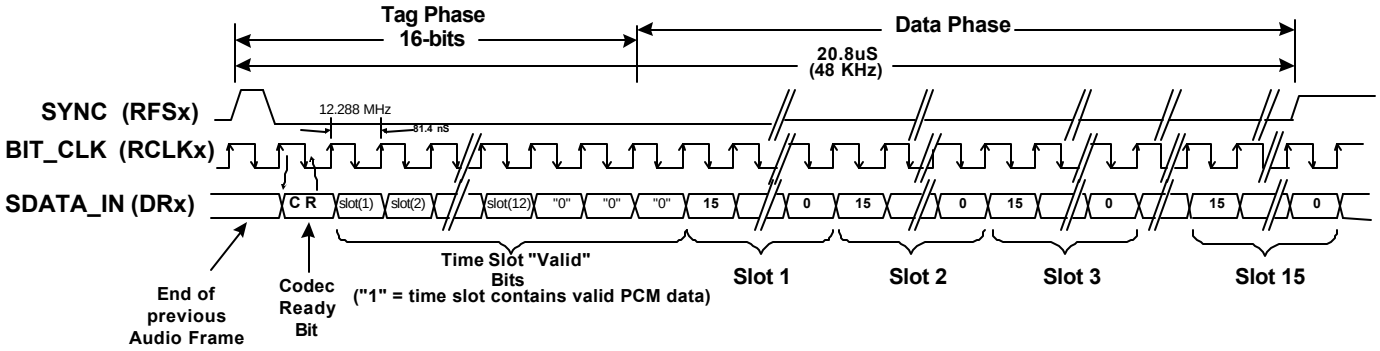


Figure 6. Modified AC-link Audio Input Frame - AD1819x to ADSP-2106x Data Path **

**Note: The timing in figure 6 differs from the standard AC-97 timing of 12 slots, 20 bits in length for the data phase portion of the audio frame. The timeslots are set to 16 bits in length (SLOT16 Mode) by the ADSP-2106x during initial enabling of the DSP SPORT so that proper data alignment of TDM slots to occur. Also, the frame sync generated by the DSP is not set for the duration of the Tag Phase, as described in the AC-97 spec and AD1819 data sheet. The DSP generates a frame sync for approximately 1 serial clock cycle. However this does not affect the codec operation, since the codec samples the frame sync only for the first cycle prior to transmission of the MSB of the Tag Phase timeslot. Setup and hold times on the AD1819 are relaxed enough to meet the SHARC RFSx external generation timings listed in the ADSP-2106x data sheet.

The audio input frame, shown in Figure 7, (data samples sent to the DSP from the AD1819x) begins with a low to high transition of SYNC/RFSx. SYNC is synchronous to the rising edge of BIT_CLK/RCLKx. On the immediately following falling edge of BIT_CLK, the AD1819x samples the assertion of SYNC. This falling edge marks the time when both sides of serial link are aware of the start of a new audio frame. On the next rising of BIT_CLK, the AD1819x transitions SDATA_IN into the first bit position of slot 0 **("Codec Ready" bit)**. Each new bit position is presented to AC-link on a rising edge of BIT_CLK, and subsequently sampled by the ADSP-2106x on the following falling edge of BIT_CLK. This sequence ensures that data transitions, and subsequent sample points for both incoming and outgoing data streams are time-aligned. The SDATA_IN's composite stream is MSB justified (MSB first) with all non-valid bit positions (for assigned and/or unassigned time slots) stuffed with 0's by the AD1819x. SDATA_IN data is sampled on the falling edges of BIT_CLK .

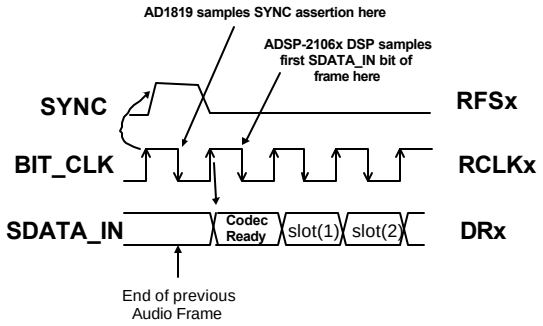


Figure 7. Start of an Audio Input Frame

2.3 Codec Ready Bit (Most Significant Bit In Slot 0)

Within slot 0 the first bit is a global bit (SDATA_IN slot 0, bit 15) which flags whether AD1819x is in the **"Codec Ready"** state or not. If the "Codec Ready" bit is a 0, this indicates that AD1819x is not ready for normal operation. This condition is normal following the deassertion of power on reset for example, while the AD1819's voltage references settle. When "Codec Ready" is a 1 it indicates that the serial-link, the AD1819 control registers, and at least one of the subsystems described in the Powerdown Control/Status Register is operational.

Prior to any attempts at putting the codec into operation the ADSP-2106x should poll the first bit in the audio input frame (SDATA_IN slot 0, bit 15) for an indication that the AD1819x has gone "Codec Ready". Below is example ADSP-2106x Assembly Language Instructions to accomplish the 'Poll Codec Ready' task:

```
Wait_Codec_Ready:                    /* Wait for CODEC Ready State */
       R0 = DM(rx_buf + 0);          /* get bit 15 status bit from AD1819 tag phase slot0*/
       R1 = 0x8000;                  /* mask out codec ready bit in tag phase */
       R0 = R0 AND R1;               /* test the codec ready status flag bit */
       IF EQ JUMP Wait_Codec_Ready;  /* if flag is lo, continue to wait for a hi */
```

Once the AD1819x is sampled "Codec Ready" then the next 15 bit positions (in Slot-16) sampled by the ADSP-2106x indicate which of the corresponding 15 time slots are assigned to input data streams, and that they contain valid data. There are several sub-functions within AD1819x that can independently go busy/ready. The global "Codec Ready" bit indicates that at least one of these sub-functions is available. It is the responsibility of the DSP to probe more deeply into the AD1819x register file to determine which AD1819x subsections are actually ready.

In addition to polling the "Codec Ready" indicator bit, the Power-Down Control/Status Register is useful for monitoring subsystem readiness. The DSP programmer can choose to poll the Power-Down Control/Status Register to wait for Reference Voltage, Analog Mixer, DAC & ADC Section Stabilization after polling the Codec Ready Bit. This step would ensure that the DSP will not modify codec registers until the conversion resources and analog circuitry have stabilized. This step is recommended by the AC-97 1.03 specification, although the source code example in Appendix A does not perform this step, since it was found the successful programming of the AD1819x was achieved after simply polling the "Codec Ready" indicator bit.

3. Configuring The ADSP-21065L Serial Port MCM Interface

When interfacing the AD1819A codec to an ADSP-21065L SHARC processor, the interconnection between the 2 devices can be through **either** SPORT0 or SPORT1. In the application code section of this document, SPORT1 is used in the example drivers since the 21065L EZ-LAB makes use of SPORT1 for the codec interface.

Both the DSP and codec serial port shift data MSB first, and the AD1819A's BIT_CLK frequency of 12.288 Mhz is less than the SCLK maximum of 40 MHz for the 2106x. Therefore, the DSP's CLKOUT frequency must be greater than 12.288 Mhz.

Figure 8. ADSP-21065L SPORTs

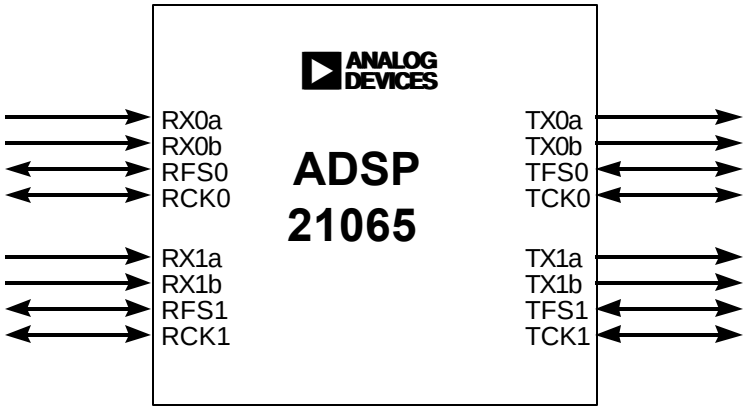


Table 1. ADSP-21065L Serial Port Pins

Function	SPORT0		SPORT1	
	A Chn	B Chn	A Chn	B Chn
Transmit data	DT0A	DT0B	DT1A	DT1B
Transmit clock	TCLK0		TCLK1	
Transmit frame sync/ word select	TFS0		TFS1	
Receive data	DR0A	DR0B	DR1A	DR1B
Receive cock	RCLK0		RCLK1	
Receive frame sync	RFS0		RFS1	

The ADSP-21065L Serial Ports have two transmit and receive data pins for both the transmit side and the receive side.

- Transmit A Channels - DT0A, DT1A
- Transmit B Channels – DT0B, DT1B
- Receive A Channels – DR0A, DR1A
- Receive B Channels – DR0B, DR1B

NOTE: The ADSP-21065L SPORT channel B pins are not functional for multichannel mode. Both the transmitter and receiver have their own serial clocks. The TFSx frame sync becomes an output 'transmit data valid' pin and is not used, while RFSx is used to control the start of a multichannel frame for both data transmission and reception.

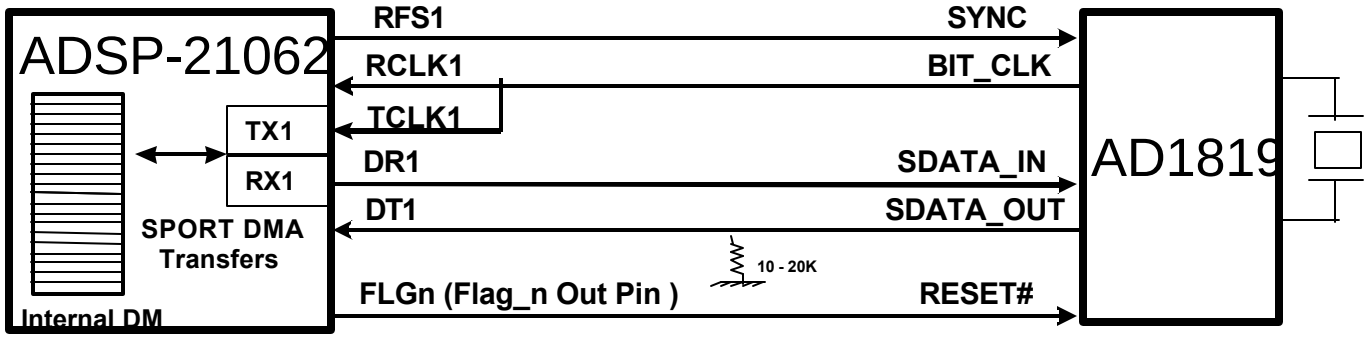


Figure 9. Example AD1819A/ADSP-21062 SHARC Serial Port 1 Interconnections (assuming 5V I/0)

IMPORTANT SERIAL INTERCONNECTION NOTES:

- *The 21065L's TFSx line is an output pin in multichannel mode (TDV - Transmit Data Valid). It should be left **unconnected** and not tied with RFSx together to the AD1819x Frame Sync. The RFSx pin is used to signal the start of a TDM frames for both reception and transmission of data. Connecting TFSx(TDVx) could cause contention with the RFSx (SYNC) and will most likely lock up the SPORT and possibly damage the RFSx pin over time!!!*

- *A 10-K to 20-K Ohm pull-down resister is recommended on the ADSP-2106x's DTx (DT0 / DT0A / DT1 / DT1A) line. The DTx lines on ADI's SHARC DSPs have a 50K internal pull-up, which can cause the AD1819A to enter a test mode, referred to in the AC'97 spec. as 'ATE Factory Test Mode'. The pull-down is required to ensure proper codec serial operation.*

- *Since BIT_CLK is the master serial clock, the DSP's RCLKx and TCLKx signals are set up for external generation, since they are slave (input) signals. To synchronize shifting of data channels, the RCLKx and TCLKx pins are tied together to BIT_CLK.*

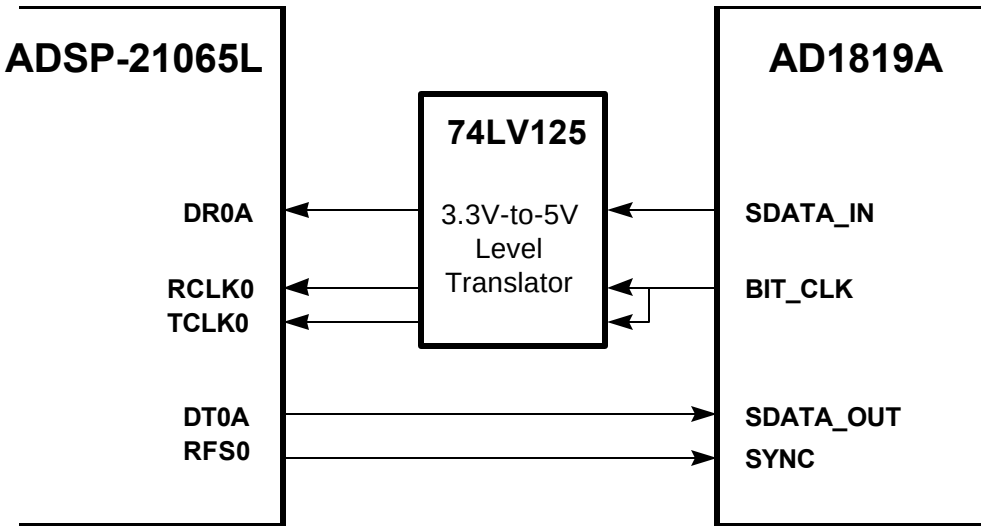
- *For 3.3 Volt Interfaces, the AD1819x output signals connected as inputs to the DSP must be level-converted down from 5.0 Volts to 3.3 Volts (See Next Section 3.1).*

3.1 ADSP-21065L - 3.3 V Level Shifting Considerations

The ADSP-21065L is a new derivative of the SHARC family that is targeted for low-cost/high-performance consumer oriented applications. Since it is a 3.3 Volt part, the 5 Volt AC-link signals that the AD1819A provides will damage the driver pins on the 21065L serial port. Level-shifting of all input signals is recommended. All SPORT output signals that are inputs to the AD1819 do not need to be level shifted since the AD1819A will recognize 3.3 volts as a valid TTL high level. Also, all other 3.3 V SHARC processors like the ADSP-21060L, ADSP-21062L and ADSP-21061L should level shift all input serial port signals.

Figure 10 below shows the interface between the AD1819A and the ADSP-21065L. ADI's new code and pin compatible AC-97 rev 2.01 parts, The AD1881/AD1881A/AD1882/AD1885, include 3.3 V digital I/O, removing the need for level shifting.

Figure 10. 21065L EZ-LAB DSP/Codec Interface



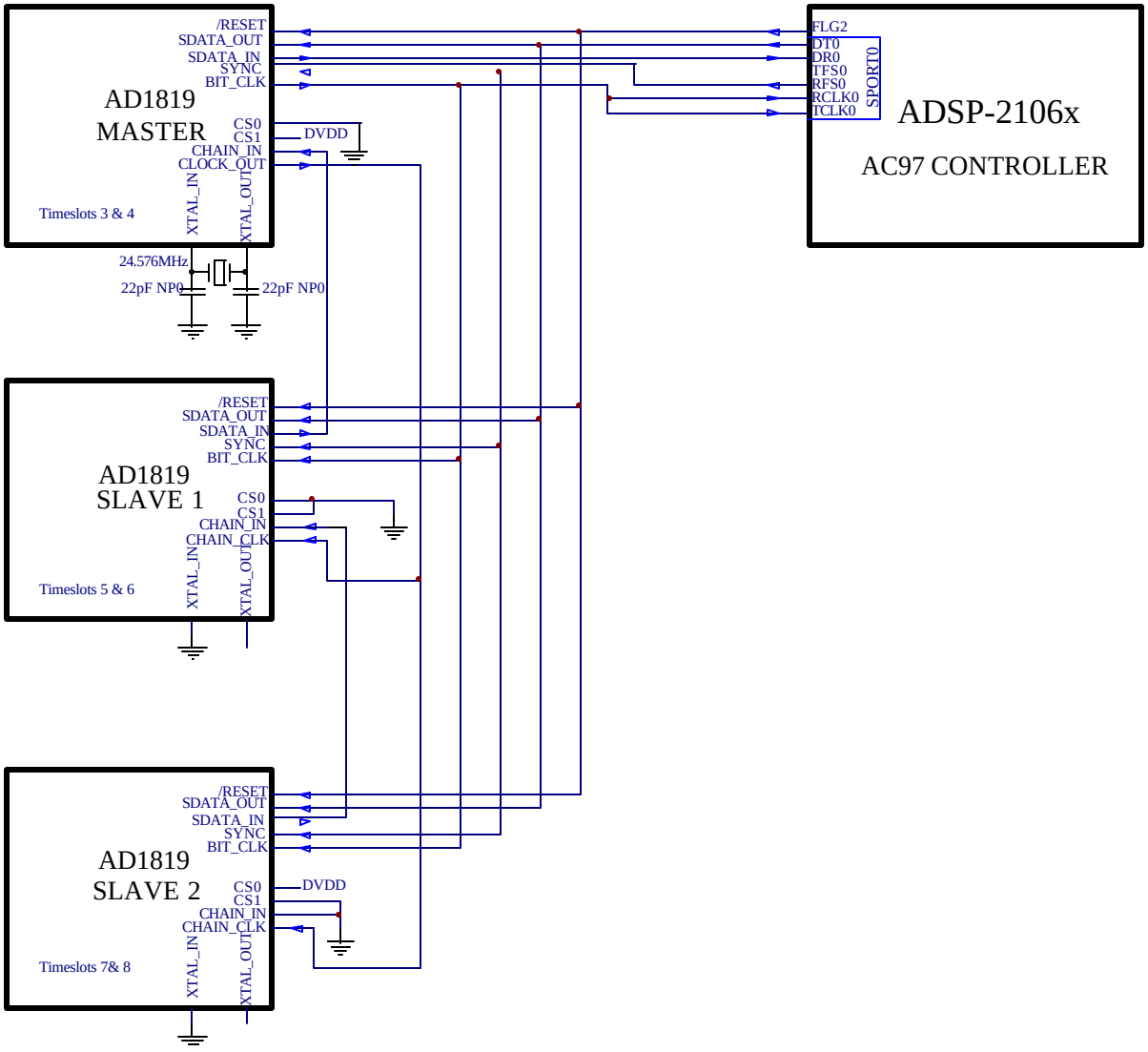
Note: *The Second Generation pin-for-pin compatible AC-97 codecs, the AD188x series, have 3.3 Volt I/O capability on the digital portion of the chip, thus level shifting is not required*

In order to facilitate serial communications with the AD1819A, the SPORT1 pin connections are configured as shown in Table 1 and Figure 9:

Table 1.

ADSP-21065L Pin:	*AD1819A Pin:*	*Driven By:*
RCLK1, TCLK1	*BIT_CLK*	*codec*
RFS1	*SYNC*	*DSP*
TFS1 (unconnected)	----------	----------
DR1A	*SDATA_IN*	*codec*
DT1A	*STATA_OUT*	*DSP*

3.2 Figure 11. Block Diagram Of A 5V ADSP-2106x Serial Interface To 3 AD1819 Codecs



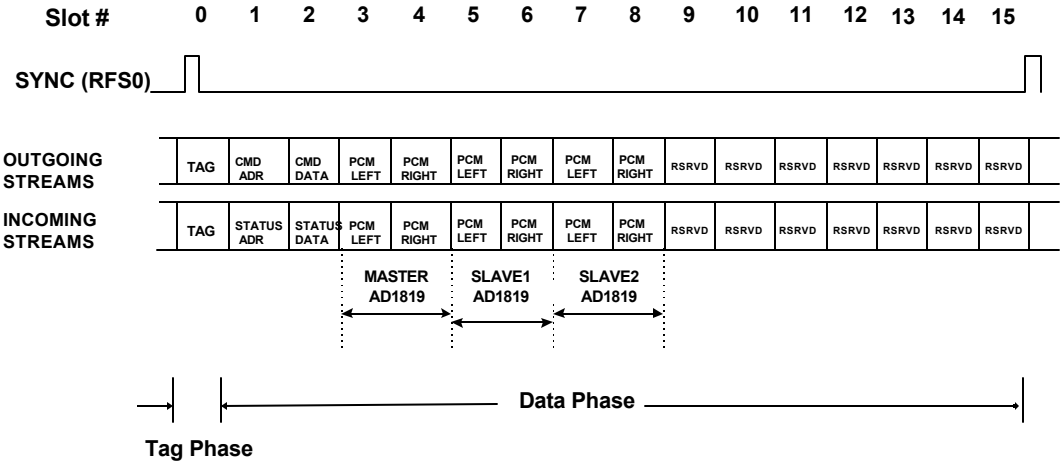


Figure 12. Timeslot Allocation For Multiple AD1819 Bi-directional TDM Audio Frame

3.3 SPORT DMA Channels And Interrupt Vectors

There are 8 dedicated DMA channels for both SPORT0 and SPORT1 on the ADSP-21065L. The IOP addresses for the DMA registers are shown in the table below for each corresponding channel and SPORT data buffer. In multichannel mode, only channels 0, 2, 4 and 6 are active, because the channel B pins are disabled in Multichannel Mode.

Table 6. 8 SPORT DMA channels and data buffers

Chn	Data Buffer	Address	Description
0	Rx0A	0x0060 0x0064	Serial port 0 receive; A data
1	Rx0B	0x0030 0x0034	Serial port 0 receive; B data
2	Rx1A	0x0068 0x006C	Serial port 1 receive; A data
3	Rx1B	0x0038 0x003C	Serial port 1 receive; B data
4	Tx0A	0x0070 0x0074	Serial port 0 transmit; A data
5	Tx0B	0x0050 0x0054	Serial port 0 transmit; B data
6	Tx1A	0x0078 0x005C	Serial port 1 transmit; B data
7	Tx1B	0x0058 0x005C	Serial port 1 transmit; B data

Each serial port has a transmit DMA interrupt and a receive DMA interrupt (shown in Table 7 below). With serial port DMA disabled, interrupts occur on a word by word basis, when one word is transmitted or received. Table 7 also shows the interrupt priority, because of their relative location to one another in the interrupt vector table. The lower the interrupt vector address, the higher priority the interrupt. Note that channels A and B for the transmit and receive side of each SPORT share the same interrupt location. Thus, data for both DMA buffers is processed at the same time, or on a conditional basis depending on the state of the buffer status bits in the SPORT control registers.

Table 7. ADSP-21065L Serial Port Interrupts

Interrupt[1]	Function	Priority
SPR0I	SPORT0 receive DMA channels 0 and 1	Highest
SPR1I	SPORT1 receive DMA channels 2 and 3	
SPT0I	SPORT0 transmit DMA channels 4 and 5	
SPT1I	SPORT1 transmit DMA channels 6 and 7	
EP0I	Ext. port buffer 0 DMA channel 8	
EP1II	Ext. port buffer 1 DMA channel 9	Lowest

[1] Interrupt names are defined in the def21065.h include file supplied with the ADSP-21000 Family Visual DSP Development Software.

3.4 Serial Port Related IOP Registers

This section briefly highlights the list of available SPORT-related IOP registers that will need to be programmed when configuring the SPORTs for Multichannel Mode. To program these registers, write to the appropriate address in memory using the symbolic macro definitions supplied in the `def21065l.h` file (included with the Visual DSP tools in the `/INCLUDE/` directory). External devices such as another 21065L, or a host processor, can write and read the SPORT control registers to set up a serial port DMA operation or to enable a particular SPORT. These registers are shown in the table below. The SPORT DMA IOP registers are covered in section 4.8. As we will see in the next section, many of the available registers shown below need to be programmed to set up Multichannel Mode. These registers are highlighted in bold text.

Table 8. Serial Port IOP Registers

	Register	IOP Address	Description
SPORT0	**STCTL0**	0xe0	SPORT0 transmit control register
	SRCTL0	0xe1	SPORT0 receive control register
	TDIV0	0xe4	SPORT0 transmit divisor
	RDIV0	0xe6	SPORT0 receive divisor
	MTCS0	0xe8	SPORT0 multichannel transmit select
	MRCS0	0xe9	SPORT0 multichannel receive select
	MTCCS0	0xea	SPORT0 multichannel transmit compand select
	MRCCS0	0xeb	SPORT0 multichannel receive compand select
	KEYWD0	0xec	SPORT0 receive comparison register
	IMASK0	0xed	SPORT0 receive comparison mask register
SPORT1	**STCTL1**	0xf0	SPORT1 transmit control register
	SRCTL1	0xf1	SPORT1 receive control register
	TDIV1	0xf4	SPORT1 transmit divisor
	RDIV1	0xf6	SPORT1 receive divisor
	MTCS1	0xf8	SPORT1 multichannel transmit select
	MRCS1	0xf9	SPORT1 multichannel receive select
	MTCCS1	0xfa	SPORT1 multichannel transmit compand select
	MRCCS1	0xfb	SPORT1 multichannel receive compand select
	KEYWD1	0xfc	SPORT1 receive comparison register
	IMASK1	0xfd	SPORT1 receive comparison mask register

In Multichannel Mode, the available SPORT data buffers that are active are the channel A registers (which are highlighted below). It is these registers that are actually used to transfer data between the AD1819A and the DMA controller on the ADSP-21065L. The DMA controller is used to transfer data to and from internal memory without any intervention from the core.

SPORT Data Buffers	**TX0_A**	0xe2	SPORT0 transmit data buffer, channel A data
	RX0_A	0xe3	SPORT0 receive data buffer, channel A data
	TX1_A	0xf2	SPORT1 transmit data buffer, channel A data
	RX1_A	0xf3	SPORT1 receive data buffer, channel A data
	TX0_B	0xee	SPORT0 transmit data buffer, channel B data
	RX0_B	0xef	SPORT0 receive data buffer, channel B data
	TX1_B	0xfe	SPORT1 transmit data buffer, channel B data
	RX1_B	0xff	SPORT1 receive data buffer, channel B data

3.5 Example SPORT1 IOP Register Configuration For Audio Processing At 48 kHz

The configuration for SPORT1, for use with the ADSP-21065L EZ-LAB at a fixed 48 kHz sample rate, is set up as follows:

- 16-bit serial word length
- Enable SPORT1 transmit and receive DMA functionality
- Enable DMA chaining functionality for SPORT1 transmit and receive
- External Serial Clock (RCLK1) - the codec provides the serial clock to the ADSP-21065L.
- Transmit and Receive DMA chaining enabled. The DSP program declares 2 buffers - **tx_buf[5]** and **rx_buf[5]** - for DMA transfers of SPORT0 transmit and receive serial data. Both buffers reserve 5 locations in memory to reflect the AD1819A time slot allocation for a single codec. DMA chaining is almost certainly required, or the interrupt service overhead will chew up too much of the DSP's bandwidth.
- Multichannel Frame Delay = 1, i.e., the frame sync occurs 1 SCLK cycle before MSB of 1st word. New frames are marked by a HI pulse driven out on SYNC one serial clock period before the frame begins.

```
Program_SPORT1_Registers:
      /* program sport0 receive control register */
      R0 = 0x0F0C40F0;           /* 16 chans, int rfs, ext rclk, slen = 15, sden&schen enabled*/
      dm(SRCTL1) = R0;           /* sport 0 receive control register */
      /* sport1 transmit control register */
      R0 = 0x001C00F0;           /* 1 cyc mfd, data depend, slen = 15, sden & schen enabled */
      dm(STCTL1) = R0;           /* sport 0 transmit control register */
```

- The ADSP-21065L provides an internally generated 48 kHz frame sync (RFS1). It must be a 48 kHz frame rate since the AC97 specified frame rate of the AD1819A is 48 kHz. Since the AD1819A serial clock is 12.288 MHz, a divide factor or 256 will produce a 48 kHz internally generated frame sync.

```
      /* sport1 receive frame sync divide register */
      R0 = 0x00FF0000;           /* SCKfrq(12.288M)/RFSfrq(48.0K)-1 = 0x00FF */
      dm(RDIV1) = R0;
```

- No companding.

```
      /* sport1 transmit and receive multichannel companding enable registers */
      R0 = 0x00000000;           /* no companding */
      dm(MRCCS1) = R0;           /* no companding on receive */
      dm(MTCCS1) = R0;           /* no companding on transmit */
```

- Multichannel Mode - Length = 5 multichannel words enabled. This allows 1 AD1819A audio frame per ADSP-21065L multichannel frame.

```
      /* sport1 receive and transmit multichannel word enable registers */
      R0 = 0x0000001F;           /* enable transmit and receive channels 0-4 */
      dm(MRCS1) = R0;
      dm(MTCS1) = R0;
```

3.6 DMA Registers For The ADSP-21065L Serial Ports 0 and 1

The following register descriptions are provided in the defs21065l.h file for programming the DMA registers associated with the I/O processor's DMA controller. We will look at how these registers are programmed for DMA chaining, in which the DMA registers are reinitialized automatically whenever a serial port interrupt request is generated in the next section.

Table 9. SPORT DMA IOP Registers

	DMA Register Description	DMA Register	IOP Address
SPORT0 Receive Channel A	DMA Channel 0 Index Register	IIR0A	0x60
	DMA Channel 0 Modify Register	IMR0A	0x61
	DMA Channel 0 Count Register	CR0A	0x62
	DMA Channel 0 Chain Pointer Register	CPR0A	0x63
	DMA Channel 0 General Purpose Register	GPR0A	0x64
SPORT0 Receive Channel B	DMA Channel 1 Index Register	IIR0B	0x30
	DMA Channel 1 Modify Register	IMR0B	0x31
	DMA Channel 1 Count Register	CR0B	0x32
	DMA Channel 1 Chain Pointer Register	CPR0B	0x33
	DMA Channel 1 General Purpose Register	GPR0B	0x34
SPORT1 Receive Channel A	DMA Channel 2 Index Register	IIR1A	0x68
	DMA Channel 2 Modify Register	IMR1A	0x69
	DMA Channel 2 Count Register	CR1A	0x6A
	DMA Channel 2 Chain Pointer Register	CPR1A	0x6B
	DMA Channel 2 General Purpose Register	GPR1A	0x6C
SPORT1 Receive Channel B	DMA Channel 3 Index Register	IIR1B	0x38
	DMA Channel 3 Modify Register	IMR1B	0x39
	DMA Channel 3 Count Register	CR1B	0x3A
	DMA Channel 3 Chain Pointer Register	CPR1B	0x3B
	DMA Channel 3 General Purpose Register	GPR1B	0x3C
SPORT0 Transmit Channel A	DMA Channel 4 Index Register	IIT0A	0x70
	DMA Channel 4 Modify Register	IMT0A	0x71
	DMA Channel 4 Count Register	CT0A	0x72
	DMA Channel 4 Chain Pointer Register	CPT0A	0x73
	DMA Channel 4 General Purpose Register	GPT0A	0x74
SPORT0 Transmit Channel B	DMA Channel 5 Index Register	IIT0B	0x50
	DMA Channel 5 Modify Register	IMT0B	0x51
	DMA Channel 5 Count Register	CT0B	0x52
	DMA Channel 5 Chain Pointer Register	CPT0B	0x53
	DMA Channel 5 General Purpose Register	GPT0B	0x54
SPORT1 Transmit Channel A	DMA Channel 6 Index Register	IIT1A	0x78
	DMA Channel 6 Modify Register	IMT1A	0x79
	DMA Channel 6 Count Register	CT1A	0x7A
	DMA Channel 6 Chain Pointer Register	CPT1A	0x7B
	DMA Channel 6 General Purpose Register	GPT1A	0x7C
SPORT1 Transmit Channel B	DMA Channel 7 Index Register	IIT1B	0x58
	DMA Channel 7 Modify Register	IMT1B	0x59
	DMA Channel 7 Count Register	CT1B	0x5A
	DMA Channel 7 Chain Pointer Register	CPT1B	0x5B
	DMA Channel 7 General Purpose Register	GPT1B	0x5C

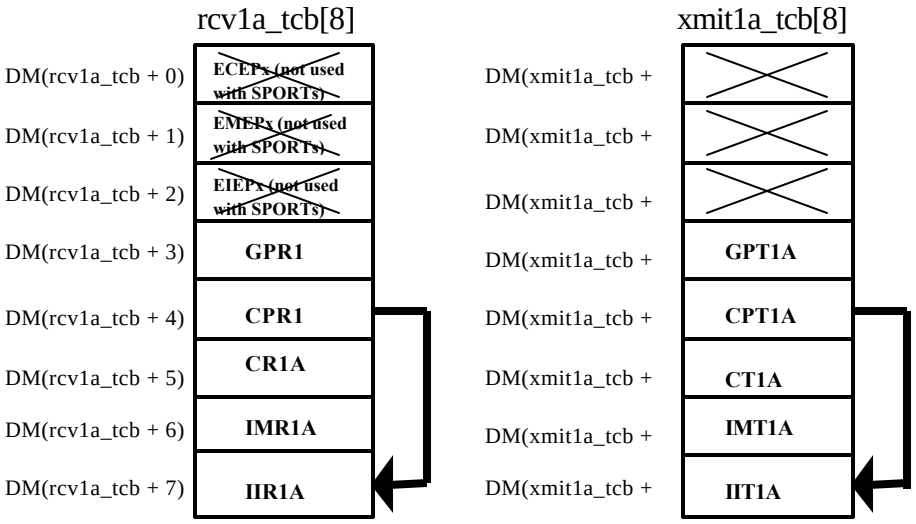
3.7 Setting Up The ADSP-21065L DMA Controller For Chained SPORT DMA Transfers

To efficiently transmit and receive digital audio data to/from the AD1819A, the recommended method is to use serial port **DMA Chaining** to transfer data between the serial bus and the DSP core. There are obvious benefits for doing this. First of all, DMA transfers allow efficient transfer of data between the serial port circuitry and DSP internal memory with zero-overhead, i.e. there is no processor intervention of the SHARC core to manually transfer the data. Secondly***, there is a one-to-one correspondence of the location of the word in the transmit and receive SPORT DMA buffers with the actual TDM audio frame timeslot on the serial bus***. Thirdly, an entire block (or audio frame) of data can be transmitted or received before generating a single interrupt. The 'chained-DMA' method of serial port processing is more efficient for the SHARC to process data, versus interrupt driven transfers, which occur more frequently, for every serial word transmitted or received. Using chained DMA transfers allows the ADSP-21065L DMA controller to autoinitialize itself between multiple DMA transfers. When the entire contents of the current SPORT buffers **rx_buf** and **tx_buf** have been received or transmitted, the ADSP-21065L can automatically set up another serial port DMA transfer that is continuously repeated for every DMA interrupt. For further information on DMA chaining, the reader can refer to section 6.3.4 in the ADSP-2106x User's Manual, or section the DMA chapter of the ADSP-21065L User's Manual.

The chain pointer register (CPxxx) is used to point to the next set of TX and RX buffer DMA chaining parameters stored in memory. SPORT DMA transfers for the AD1819A are initiated by writing the DMA buffer's memory address to the **CPR1A** register for SPORT1 receive and **CPT1A** register for SPORT1 transmit. The transmit and receive **SCHEN_A** and **SCHEN_B** bits in the SPORTx Control registers enable DMA chaining.

To auto-initialize repetitive DMA-chained transfers, the programmer needs to set up a buffer in memory called a **Transfer Control Block (TCB**) that will be used to initialize and further continue the chained DMA process. Transfer Control Blocks are locations in Internal Memory that store DMA register information in a specified order. For example, Figure 13 below demonstrates defined TCBs in internal memory for SPORT1 Channel A. The **Chain Pointer Register** (**CPR1A** and **CPT1A**) stores the location of the next set of TCB parameters to be automatically be downloaded by the DMA controller at the completion of the DMA transfer, which in this case it points back to itself to repeat the same ‘

Figure 13. TCBs for Chained DMA Transfers of SPORT1 Channel A Receive and Transmit



The TCBs for both the transmit and receive buffers are can be defined in the variable declaration section of the DSP assembly or C code. In the AD1819A initialization code shown in appendix A, the TCBs for SPORT1 channel A are defined as follows:

```
.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* receive tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 5, 1, 0;      /* transmit tcb */
```

Note that the DMA count and modify values can be initialized in the buffer declaration so that they are resident after a DSP reset and boot. However, at runtime, further modification of the buffer is required to initiate the DMA autobuffer process.

To setup and initiate a chain of SPORT DMA operations at runtime, the 21065L program can follow this sequence:

1. Set up SPORT transmit and Receive TCBs (transfer control blocks). The TCBs are defined in the data variable declaration section of your code. Before setting up the values in the TCB and kicking off the DMA process, make sure the SPORT registers are programmed along with the appropriate chaining bits required in step 2.

2. Write to the SPORT0 transmit and receive control registers (STCTL0 and STCRL0), setting the **SDEN_A** enable bit to 1 and the **SCHEN_A** chaining enable bit to a 1.

3. Write the internal memory index address register (IIxxx) of the first TCB to the CPxxx register to start the chain. The order should be as follows:
    a) write the starting address of the SPORT DMA buffer to the TCBs **internal index register IIxxx** location (TCB buffer base address + 7). You need to get the starting address of the defined DMA buffer at runtime and copy it into this location in the TCB.
    b) write the **DMA internal modify register** value **IMxxx** to the TCB (TCB buffer base address + 6). Note that this step may be skipped if it the location in the buffer was initialized in the variable declaration section of your code.
    c) write the **DMA count register Cxxx** value to the TCB (TCB buffer base address + 5). Also note that this step may be skipped if it the location in the buffer was initialized in the variable declaration section of your code.
    d) get the IIxxx value of the TCB buffer that was previously stored in step (a), **set the PCI bit** with a that internal address value, and write the modified value to the chain pointer location in the TCB (TCB buffer base offset + 4).
    e) write the same 'PCI-bit-set' internal address value from step (d) manually into that DMA channel's **chain pointer register (CPxxx)**. At this moment the DMA chaining begins.

The DMA interrupt request occurs whenever the Count Register decrements to zero.

SPORT DMA chaining occurs independently for the transmit and receive channels of the serial port. After the SPORT1 receive buffer (*rx_buf*) is filled with new data, a SPORT1 receive interrupt is generated, and the data placed in the receive buffer is available for processing. The DMA controller will autoinitialize itself with the parameters set in the TCB buffer and begin to refill the receive DMA buffer with new data in the next audio frame. The processed data is then placed in the SPORT transmit buffer, where it will then be DMA'ed out from memory to the SPORT DT1A pin. After the entire buffer is transmitted from internal memory to the SPORT circuitry, the DMA controller will autoinitialize itself with the stored TCB parameters to perform another DMA transfer of new data that will be placed in the same transmit buffer (*tx_buf)*.

Below are example assembly instructions used to set up the receive and transmit DMA buffers and Transfer Control Blocks for SPORT1 Channel A, which is shown in the 21065L EZ-LAB example shown in appendix A. These values are reloaded from internal memory to the DMA controller after the entire SPORT DMA buffer has been received or transmitted.

```
.segment /dm    dm_codec;

/* define DMA buffer sizes to match number of active TDM channels */
.var rx_buf[5];      /* receive buffer */

                     /* transmit buffer */
.var tx_buf[5] = ENABLE_VFbit_SLOT1_SLOT2,      /* set valid bits for slot 0, 1, and 2 */
                 SERIAL_CONFIGURATION,          /* serial configuration register address */
                 0xFF80,                     /* set to slot-16 mode for ADI SPORT compatibility */
                 0x0000,                        /* stuff other slots with zeros for now */
                 0x0000;

/* DMA Chaining Transfer Control Blocks */
.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* receive tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 5, 1, 0;      /* transmit tcb */

.endseg;
```

```
.segment /pm  pm_code;
/*-------------------------------------------------------------------------*/
/*            DMA Controller Programming For SPORT1 MCM Tx and Rx           */
/*                                                                          */
/*            Setup SPORT1 for DMA Chaining:                                */
/*-------------------------------------------------------------------------*/

Program_DMA_Controller:
       r1 = 0x0001FFFF;           /* cpx register mask */
       /* sport1 dma control tx chain pointer register */
       r0 = tx_buf;
       dm(xmit_tcb + 7) = r0;     /* internal dma address used for chaining */
       r0 = 1;
       dm(xmit_tcb + 6) = r0;     /* DMA internal memory DMA modifier */
       r0 = 5;
       dm(xmit_tcb + 5) = r0;     /* DMA internal memory buffer count */
       r0 = xmit_tcb + 7;         /* get DMA chain intn mem pointer containing tx_buf address */
       r0 = r1 AND r0;            /* mask the pointer */
       r0 = BSET r0 BY 17;        /* set the pci bit */
       dm(xmit_tcb + 4) = r0;     /* write DMA transmit block chain pointer to TCB buffer */
       dm(CPT1A) = r0;            /* transmit block chain pointer, initiate tx0 DMA transfers */
       /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
       /* - Note: Tshift0 & TX0 will be automatically loaded with the first 2 values in the   -*/
       /* - Tx buffer. The Tx buffer pointer ( IIT1A ) will increment twice by the modify     -*/
       /* - modify value specified in ( IMT1A ).                                             -*/
       /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

       /* sport1 dma control rx chain pointer register */
       r0 = rx_buf;
       dm(rcv_tcb + 7) = r0;      /* internal dma address used for chaining */
       r0 = 1;
       dm(rcv_tcb + 6) = r0;      /* DMA internal memory DMA modifier */
       r0 = 5;
       dm(rcv_tcb + 5) = r0;      /* DMA internal memory buffer count */
       r0 = rcv_tcb + 7;          /* get DMA chain intn mem pointer containing rx_buf address */
       r0 = r1 AND r0;            /* mask the pointer */
       r0 = BSET r0 BY 17;        /* set the pci bit */
       dm(rcv_tcb + 4) = r0;      /* write DMA receive block chain pointer to TCB buffer */
       dm(CPR1A) = r0;            /* receive block chain pointer, initiate rx0 DMA transfers */

.endseg;
```

3.8 AD1819A TDM Serial Port Time Slot Assignments, DMA Buffer Relationships

The DSP SPORT Multichannel Mode Time Slot Map for AD1819A communication in ***SLOT16 Mode*** is as follows:

Timeslot	SDATA_OUT Pin (DT1A)	SDATA_IN Pin (DR1A)
0	Tag Phase (ADSP-2106x)	Tag Phase (Codec)
1	Command Address Port (Control Word Input)	Status Address Port (Status Word Output)
2	Command Data Port (Control Register Data Input)	Status Data Port (Control Register Read Data Output)
3	Master PCM Playback Left Channel	Master PCM Capture (Record) Left Channel
4	Master PCM Playback Right Channel	Master PCM Capture Right Channel
5	Slave 1 PCM Playback Left Channel	Slave 1 PCM Capture Left Channel
6	Slave 1 PCM Playback Right Channel	Slave 1 PCM Capture Right Channel
7	Slave 2 PCM Playback Left Channel	Slave 2 PCM Capture Left Channel
8	Slave 2 PCM Playback Right Channel	Slave 2 PCM Capture Right Channel
9	Reserved for Future Use (should always stuff with 0s)	Reserved for Future Use (AD1819 fills with 0s)
10	Reserved for Future Use (should always stuff with 0s)	Reserved for Future Use (AD1819 fills with 0s)
11	Reserved for Future Use (should always stuff with 0s)	Reserved for Future Use (AD1819 fills with 0s)
12	Reserved for Future Use (should always stuff with 0s)	Reserved for Future Use (AD1819 fills with 0s)
13	Reserved Slot, SLOT16 Mode extension	Reserved Slot, SLOT16 Mode extension
14	Reserved Slot, SLOT16 Mode extension	Reserved Slot, SLOT16 Mode extension
15	Reserved Slot, SLOT16 Mode extension	Reserved Slot, SLOT16 Mode extension

Corresponding ADSP-21065L SPORT0 DMA Buffer Addresses For Associated Timeslots

rx_buf[9] - DSP SPORT DMA receive buffer

Slot #	*Description*	*DSP Data Memory Direct Address*
0	Tag Phase (AD1819)	DM(rx_buf + 0)
1	Status Address Port	DM(rx_buf + 1)
2	Status Data Port	DM(rx_buf + 2)
3	Master PCM Capture (Record) Left Channel	DM(rx_buf + 3)
4	Master PCM Capture Right Channel	DM(rx_buf + 4)
5	Slave 1 PCM Capture Left Channel	DM(rx_buf + 5)
6	Slave 1 PCM Capture Right Channel	DM(rx_buf + 6)
7	Slave 2 PCM Capture Left Channel	DM(rx_buf + 7)
8	Slave 2 PCM Capture Right Channel	DM(rx_buf + 8)

tx_buf[9] - DSP SPORT DMA transmit buffer

Slot #	*Description*	*DSP Data Memory Direct Address*
0	Tag Phase (DSP)	DM(tx_buf + 0)
1	Command Address Port	DM(tx_buf + 1)
2	Command Data Port	DM(tx_buf +1)
3	Master PCM Playback Left Channel	DM(tx_buf +2)
4	Master PCM Playback Right Channel	DM(tx_buf +3)
5	Slave 1 PCM Playback Left Channel	DM(tx_buf +4)
6	Slave 1 PCM Playback Right Channel	DM(tx_buf +5)
7	Slave 2 PCM Playback Left Channel	DM(tx_buf +6)
8	Slave 2 PCM Playback Right Channel	DM(tx_buf +7)

Note: Even though there are 16 slots in the audio frame, the DMA buffer size (as well as the number of channels enabled in the SPORT multichannel control registers) should be set to the size of the number of slots containing valid data to reduce IOP-bus overhead. For a single codec system, the buffer sizes should be 5 words. For a dual codec system, the buffer sizes should be 7 words while for a triple codec system, the DMA buffers are set to 9 words in length. However, when processing data from the transmit interrupt while running the sample rate less than 48 kHz, it is recommended to add two dummy slots, or two dummy words to the transmit DMA buffer. For 1, 2 or 3 codecs, this would correspond to 7, 9 or 11 words. We will cover these recommendations in sections 6.1, 6.2 and 6.3.

4. The AD1819's Serial Configuration Register (Address 0x74)

The AD1819's serial configuration register (located at codec index address 0x74) has additional functionality which is an Analog Devices addition to the register mapping of the AC'97 specification. Understanding this register is key for successful communication between the DSP and multiple AD1819As, especially for variable sample rate applications running less than 48 kHz. The Serial Configuration Register allows the DSP to perform the following functions:

- Operate the AC-link in **SLOT-16 mode**, all slots are 16-bits in length. This mode should be set as soon as the codecs are fully functional.
- Set **Codec Register Mask Bits** for the Master Codec, Slave 1 Codec, or Slave 2 codec, thus allowing the DSP to communicate to 1 codec at a time for setting/reading registers. Setting all 3 Mask bits will allow the DSP to program all 3 codecs at the same time.
- Set an enable bit that will force the Status Address and Data Slots (Slots 1 and 2) to display the contents of the serial configuration by default. This will allow the host processor to read the **DAC request bits** to see if the codecs are requesting data, which is required if the DACs are running at a slower rate or different rate than the ADCs.

Below is more detailed description of the Serial Configuration Register:

Serial Configuration (Index 74h)

Reg Number	Name	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Default
74h	Serial Configuration	SLOT16	REGM2	REGM1	REGM0	DRQEN	DLRQ2	DLRQ1	DLRQ0	X	X	X	X	X	DRRQ2	DRRQ1	DRRQ0	X

SLOT16 Enable 16-bit slots
SLOT16 makes all AC Link slots 16 bits in length, formatted into 16 Slots

REGM0 Master codec register mask

REGM1 Slave 1 codec register mask

REGM2 Slave 2 codec register mask

If your system uses only a single AD1819, you can ignore the register mask and the slave 1/slave 2 request bits. If you write to this register, write ones to all of the register mask bits. The DxRQx bits are read-only.

DRQEN Fill idle status slots with DAC request reads,

and stuffs DAC requests into LSB of output address slot (AC-Link Slot 1).

If you set the DRQEN bit, then the AD1819 will fill all otherwise- unused AC-link status address & data slots with the contents of register 74h. That makes it somewhat simpler to access the information, because you don't need to continually issue Aclink read commands to get the register contents. Also, the DAC requests are reflected in Slot 1, bits (11….6).

DRRQ0 Master codec DAC right request

DRRQ1 Slave 1 codec DAC right request

DRRQ2 slave 2 codec DAC right request

DLRQ0 Master codec DAC left request

DLRQ1 Slave 1 codec DAC left request

DLRQ2 Slave 2 codec DAC left request

The codec asserts the DxRQx bit when the corresponding DAC channel can accept data in the next frame. These bits are snapshots of the codec state taken when the current frame began (effectively, on the rising edge of SYNC), but they also take notice of DAC samples in the current frame.

4.1 Configuring The AD1819A Serial Link To SLOT-16 Mode For ADI SPORT Compatibility

Slot 16-Mode allows an efficient communication interface between DSPs and the AD1819A. Slot-16 mode is useful since the TAG is always 16-bits and equal length slots eases to use of serial port autobuffering or DMA chaining. DSPs that support a TDM interface usually do not provide the capability to program different slots to different word lengths. This mode ensures that all 16 slots are 16-bits, allowing a much easier interface to 16-bit/32-bit DSPs. The DSP will generate a frame sync every 256 serial clock cycles, so instead of having 1 16 bit Tag Phase slot with 12, 20-bit slots, the AD1819A will generate 16, 16-bit slots in 256 serial clock cycles:

16 bit Tag Phase + (12 x 20-bit timeslots)= 256 bit clock cycles

now becomes,

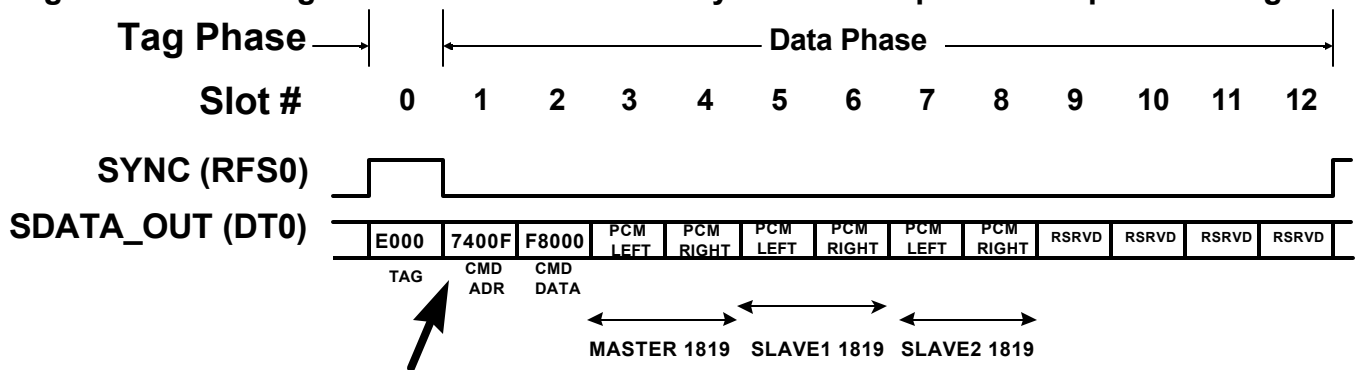
16 bit Tag Phase + (15 x 16-bit timeslots) = 256 bit clock cycles

Note that the DSP will generate a frame sync every 256 serial clock cycles. With an SCLK running at 12.288 MHz, the DSP will then produce the 48KHz frame sync in SLOT-16 mode.

To initially configure the AD1819A to conform to DSP TDM schemes, the DSP should initially program the AD1819s for 16-bit slots as soon as the codec (or multiple codecs) are operational. A successful technique that is has been used by ADI's DSP Applications Group is to initially fill up the SPORT transmit buffer with the register information to set the codecs in SLOT-16 mode. As soon as the DSP serial port operation is enabled and the codecs are reset and fully operational, the codecs will respond to the DSP's repeated request to set up the AC-link to SLOT-16 mode. For example, the 21065L DSP codec driver (shown in Appendix A) initially fills the tx_buf with the correct tag phase info, serial configuration address, and data to set the codecs to SLOT-16 mode with all 3 codec register mask bits set. The buffer initialization is shown below:

```
#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;
```

Figure 14. Enabling SLOT16 Mode Immediately After DSP Sport TDM Operation Begins



Note, 16-bit DSP data intended for slot 2 is shifted over 4-bits into slot-1 because of default 20-bit slots

After codec reset, slots 1-11 are 20-bit slots, the DSP needs to ensure that it's desired codec register data in slot 2 is shifted by 4 bits to take into account that slot 1 is 20 bits after SPORT operation is enabled (Figure 14). So, instead of writing 0xF800 into the Serial Configuration Register, the DSP sends 0xFF80. The AD1819 will then recognize the data in the 20-bit Command Register Data Slot and see that SLOT-16 mode is required, as well as enabling the register mask bits for all 3 codecs. Setting the mask bits for all 3 codecs will allow us to program all 3 codecs at the same time to the same register configuration. Once SPORT0 is enabled and DMA transfers are initialized, the DSP will start transmitting the above information to set up the codecs for 16-bits per slot.

4.2 Programming Multiple AD1819xs Via The Serial Configuration Register

As stated earlier, the Serial Configuration Register (Address 0x74) is a Vendor Defined register by Analog Devices. Multi-codec index register communication is easily manageable through the use of the REGMx bits (D12, D13 and D14) for the Master , Slave 1 and Slave 2 codecs. The 3 bits are shown in the bit-level chart of address 0x74. Setting the desired REGM bit corresponding to one of the 3 codecs will determine if that codec will respond to Command Register reads and writes.

Serial Configuration (Index 74h)

Reg Number	Name	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Default
74h	Serial Configuration	SLOT16	REGM2	REGM1	REGM0	DRQEN	DLRQ2	DLRQ1	DLRQ0	X	X	X	X	X	DRRQ2	DRRQ1	DRRQ0	X

Index 74h, Bit D12 (REGM0) enables Master AD1819 indexed address reads/writes

Index 74h, Bit D13 (REGM1) enables Slave 1 AD1819 indexed address reads/writes

Index 74h, Bit D14 (REGM2) enables Slave 2 AD1819 indexed address reads/writes

The table below shows the Mask bit selection label names used in the 2106x codec driver example. These labels can be used to set the Serial Configuration Register for accessing registers on any of the 3 codecs. Whenever command register writes are sent to address 0x74, the DSP can use 'SET BIT' instructions using these labels to set codec data bits D12, D13 and D14.

Selected AD1819	#define macro Label Name	Mask Bits	Codec Register Data Bits: D15	D14	D12	D11	D10	D11
Master	MASTER_Reg_Mask	0x1000	x	0	0	1	x	..
Slave1	SLAVE1_Reg_Mask	0x2000	x	0	1	0	x	..
Slave2	SLAVE2_Reg_Mask	0x4000	x	1	0	0	x	..
Master & Slave1	MASTER_SLAVE1	0x3000	x	0	1	1	x	..
Master & Slave2	MASTER_SLAVE2	0x5000	x	1	0	1	x	..
Broadcast To All	MASTER_SLAVE1_SLAVE2	0x7000	x	1	1	1	x	..

Note: The 21062/Triple AD1819A MAFE EZ-LAB Codec Driver assumes that all REGMx bits are set to enable broadcast data writes to all codec indexed addresses. For the 21065L EZ-LAB Single AD1819A driver, only the Master REGM0 bit is set.

Thus a write to a codec indexed command register will broadcast register data writes to all 3 codecs. Reading codec registers will result in a logical OR'ing of the index register of all masked codecs. For example, with 3 codec mask bits set, reading any given register address will result in data in all 3 registers being logically OR'ed together. When attempting a read of multiple registers at the same time, the codec higher up in the chain will take precedence. For example, when reading a register from all 3 codecs, the value of the Master's requested register contents will be transmitted on the serial bus.

4.3 The AD1819A Serial Configuration Register Master And Slave DAC Request Bits

The Serial Configuration Register (Address 0x74), a Vendor Defined register by Analog Devices, includes support for transmitting data to the DACs at different sample rates than the ADCs. For Variable Sample Rate support, Analog Devices added DAC request bits to the AD1819A's Serial Configuration Register. This feature allows sample rate conversion to be done in the AD1819A/DSP TDM interface itself, removing the burden from the DSP to have to include DSP interpolation or decimation routines to change from one sample rate to another. AD1819A Variable Sample Rate Support is defined as follows:

The AD1819A is capable of sampling analog signals or converting digital signals from 7 kHz to 48 kHz in 1 Hz increments on either the left and right ADCs and left and right DACs. Two sample rate generator registers are included in the AD1819A, and either the left or right ADC and DAC channels can be assigned to either sample rate generated to sample or convert signals at a desired sample rate. The normal AC-97 TDM protocol specifies a fixed 48 kHz sample rate, in which a valid sample is transmitted or received every audio frame. Since the AD1819A can run at slower sample rates, there will not always be a valid sample in every 48 kHz audio frame. If the application requires sample rate conversion, the DSP would need to know when a valid DAC sample is requested from the AD1819A. For example, if the ADC sample rate is different than the DAC sample rate, the DSP would need to know when to transmit DAC data only when the AD1819A needs a new DAC sample. To accomplish this, the AD1819A's Serial Conversion Register Includes Left and Right DAC request bits (for the Master, Slave1 and Slave2 AD1819s) so that it will notify the AC-97 host processor that it needs a new DAC sample in the next audio frame (based on it's modified 1-Hz increment sample rate).

Serial Configuration (Index 74h)

Reg Number	Name	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Default
74h	Serial Configuration	SLOT16	REGM2	REGM1	REGM0	DRQEN	DLRQ2	DLRQ1	DLRQ0	X	X	X	X	X	DRRQ2	DRRQ1	DRRQ0	X

To activate the DAC request bit support (which by reset default is not in operation), the DSP driver must write to bit D11, the **DRQEN** bit, in the Serial Configuration Register. Once enabled, the idle Status Address Timeslot (AC-Link Timeslot 1) will always transmit the DAC requests into the Least Significant Bits of the Address Timeslot. The previously idle Status Data Timeslot (AC-Link Timeslot 2) now will always contain the contents of the Serial Configuration Register 0x74. This makes it much easier to access the DAC request bit information, because the DSP will not have to continually issue AC-link read commands to get the DAC request bit contents from register 0x74. The DAC request bits for a single codec AD1819A system should be inspected by looking at the **DLRQ0** bit (bit D8, the Master codec DAC left request - Slot 3) for left channel sample requests and the **DRRQ0** (bit D0, the Master codec DAC right request bit - Slot 4) for right channel DAC sample requests.

The AD1819A asserts the DxRQx bit when the corresponding DAC channel can accept data in the next frame. These bits are snapshots of the codec state taken when the current frame began (effectively, on the rising edge of SYNC), but they also take notice of DAC samples in the current frame. If you choose to poll DAC requests in the Status Data Slot, these request bits are ACTIVE HIGH. If you choose to poll DAC request in the Status Address Slot (Timeslot 1) while in **SLOT-16** mode with **DRQEN** enabled, these bits are ACTIVE LOW and should be inspected in the following locations in bits **D7 - D2** (not D11 - D6, which is true if we are in AC-97 mode). The bit ordering of the DAC request bits is as follows:

Status Address Slot Bit Assingments with DRQEN enabled (in any given audio frame, 0 = Request, 1=No Request):	
Bit D15	Reserved (Stuffed with 0)
Bit D14 - D8	Control Register Index (when no codec commands in prior frame.. this always shows 0x74)
Bit D7	DLRQ0 - DAC Request Slot 3 - Master Codec Left
Bit D6	DLRQ0 - DAC Request Slot 4 - Master Codec Right
Bit D5	DLRQ1 - DAC Request Slot 5 - Slave1 Codec Left
Bit D4	DLRQ1 - DAC Request Slot 6 - Slave1 Codec Right
Bit D3	DLRQ2 - DAC Request Slot 7 - Slave2 Codec Left
Bit D2	DLRQ2 - DAC Request Slot 8 - Slave2 Codec Right
Bit D1 -D0	Reserved (Stuffed with 0s)

5. DSP Programming Of The AD1819A Indexed Control Registers

Addr.	*Index Register Name*	*#define label in 2106x program*	*Defined State*	*Set By DSP?*
0x00	Reset	REGS_RESET	0x0400	N
0x02	**Master Volume**	**MASTER_VOLUME**	**0x0000**	**Y**
0x04	Reserved	RESERVED_REG_1	0xXXXX	N
0x06	Master Volume Mono	MASTER_VOLUME_MONO	0x8000	Y
0x08	Reserved	RESERVED_REG_2	0xXXXX	N
0x0A	PC_BEEP Volume	PC_BEEP_VOLUME	0x8000	Y
0x0C	Phone Volume	PHONE_VOLUME	0x8008	Y
0x0E	**Microphone Volume**	**MIC_VOLUME**	**0x8008**	**Y**
0x10	**Line In Volume**	**LINE_IN_VOLUME**	**0x8808**	**Y**
0x12	CD Volume	CD_VOLUME	0x8808	Y
0x14	Video Volume	VIDEO_VOLUME	0x8808	Y
0x16	Aux Volume	AUX_VOLUME	0x8808	Y
0x18	PCM Out Vol	PCM_OUT_VOLUME	0x8808	Y
0x1A	**Record Select**	**RECORD_SELECT**	**0x0404**	**Y**
0x1C	**Record Gain**	**RECORD_GAIN**	**0x0F0F**	**Y**
0x1E	Reserved	RESERVED_REG_3	0xXXXX	N
0x20	**General Purpose**	**GENERAL_PURPOSE**	**0x8000**	**Y**
0x22	3D Control	THREE_D_CONTROL_REG	0x0000	Y
0x24	Reserved	RESERVED_REG_4	0xXXXX	N
0x26	**Power-Down Control/Status**	**POWER_DOWN_CNTL_STAT**	**0x000X**	**N**
0x28	Reserved	RESERVED_REG_5	0xXXXX	N
0x74	**Serial Configuration**	**SERIAL_CONFIGURATION**	**0xFF80**	**Y**
0x76	Miscellaneous Control Bits	MISC_CONTROL_BITS	0x0000	Y
0x78	**Sample Rate 0**	**SAMPLE_RATE_GENERATE_0**	**0xBB80**	**Y**
0x7A	Sample Rate 1	SAMPLE_RATE_GENERATE_1	0xBB80	Y
0x7C	Vendor ID1	VENDOR_ID1	0x4144	N
0x7E	Vendor ID2	VENDOR_ID2	0x5300	N

*** Registers highlighted in bold have been altered from their default states by the 21065L 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.*

All indexed control registers that are used are initially set by the ADSP-21065L using a DSP memory buffer, where all register addresses stored on even number memory buffer locations, and their corresponding register data stored at adjacent odd numbered memory locations in the buffer. In our ADSP-21065L example, 17 registers are programmed during codec initialization. An example assembly language buffer initialization is shown below:

```
.var Init_Codec_Registers[34] =
        MASTER_VOLUME,              0x0000,
        MASTER_VOLUME_MONO,         0x8000,
        PC_BEEP_Volume,             0x8000,
        PHONE_Volume,               0x8008,
        MIC_Volume,                 0x8008,
        LINE_IN_Volume,             0x0000,
        CD_Volume,                  0x8808,
        VIDEO_Volume,               0x8808,
        AUX_Volume,                 0x8808,
        PCM_OUT_Volume,             0x0808,
        RECORD_SELECT,              0x0404,
        RECORD_GAIN,                0x0F0F,
        GENERAL_PURPOSE,            0x8000,
        THREE_D_CONTROL_REG,        0x0000,
        MISC_CONTROL_BITS,          0x0000,
        SAMPLE_RATE_GENERATE_0,     0xBB80,
        SAMPLE_RATE_GENERATE_1,     0xBB80;
```

5.1 Programming AD1819A Registers Using A Zero Overhead Loop Construct

The following assembly language hardware DO LOOP shows how the values in the **Init_Codec_Registers[ ]** buffer are sent to the appropriate slots on the Serial Port TDM bus:

```
#define             ENABLE_Vfbit_SLOT1_SLOT2    0xE000
#define             TAG_PHASE                   0
#define             COMMAND_ADDRESS_SLOT        1
#define             COMMAND_DATA_SLOT           2

 Initialize_1819_Registers:
       I4 = Init_Codec_Registers;         /* pointer to codec initialization commands */
       r15 = ENABLE_Vfbit_SLOT1_SLOT2;    /* enable valid frame bit,and slots 1&2 val data bits */

       LCNTR = 17, DO Codec_Init UNTIL LCE;
              dm(tx_buf + TAG_PHASE) = r15 /* set valid slot bits in tag phase for slots 0,1,2 */
              r1 = dm(I4, 1);                          /* fetch next codec register address */
              dm(tx_buf + COMMAND_ADDRESS_SLOT) = r1;  /* put codec reg address into tx slot 1 */
              r1 = dm(I4, 1);                          /* fetch register data contents */
              dm(tx_buf + COMMAND_DATA_SLOT) = r1;     /*put codec register data into tx slot 2*/
Codec_Init:   idle;                                    /* wait until TDM frame is transmitted */
```

Explanation Of The AD1819A Codec Initialization Loop :

- The buffer pointer is first set to point to the top of the codec register buffer.
- Slot 0 (TAG Phase) is set up to enable the valid frame bit, slot 1 valid bit and slot 2 valid bit by writing a value of 0xE000 in DM(tx_buf + 0)
- The Loop Counter Register LCNTR is set to the number of registers to be programmed. In this case 17 registers in all three codecs will be set to the same value. For multiple codecs programmed to the same configuration, by initially filling tx_buf with initialized Serial Configuration Register Data, we set up the codecs for SLOT16 mode with all three Codec Register Mask bits set. So all three codecs will then respond to command register address and data writes.
- Memory writes to DM(tx_buf + 1) will set the codec register address
- Memory writes to DM(tx_buf + 2) will send the register write data for the codec address specified in the previous timeslot.
- The IDLE instruction will allow you to do nothing but wait for a SPORT0 transmit interrupt after data has be placed in the appropriate locations in the SPORT DMA buffer tx_buf. Waiting for the SPORT interrupt will guarantee that all data in the transmit buffer has been shifted out on the TDM bus, thus telling us it is safe to go the next codec command register address and register data value in the initialization buffer and transfer those contents in the 'transmit buffer' queue.

5.2 Readback Of AD1819A Registers For Verification And Debugging Using A Zero-Overhead DO LOOP

There may be instances during the debugging stage of driver code, the DSP programmer may want to verify the desired values of the AD1819A's internal registers. One easy way to do this is to set up an output buffer where all read requests of registers can be stored after codec initialization. The readback and status of codec registers can also be done using a hardware loop. The following assembly language instructions shown below are used to initiate codec read requests of registers shown in the **Init_Codec_Registers[ ]** buffer, which is the name of the buffer used on our AD1819a driver. The results of the read requests are then placed in an output buffer called **Codec_Init_Results[ ],** in which even DSP memory addresses contain the AD1819A register address, and the DSP's odd address in the buffer contains the register data for the AD1819A address. On the 21065L EZ-LAB, the AD1819A registers can then be verified with JTAG emulator or the VDSP RS232 debug monitor by setting a breakpoint after this section of code and opening up the memory window that shows the values stored in the memory buffer. After successful debugging of custom code these instructions can then be removed.

```
#define       ENABLE_Vfbit_SLOT1          0xC000
#define       TAG_PHASE                   0
#define       COMMAND_ADDRESS_SLOT        1
#define       STATUS_ADDRESS_SLOT         1
#define       STATUS_DATA_SLOT            2

/* Verify integrity of AD1819a indexed control register states to see if communication was
successful */
verify_reg_writes:
       I4 = Init_Codec_Registers;
       m4 = 2;
       I5 = Codec_Init_Results;
       r15 = ENABLE_Vfbit_SLOT1;         /* enable valid frame bit, and slots 1 data bits */

       LCNTR = 17, Do ad1819_register_status UNTIL LCE;
              dm(tx_buf + TAG_PHASE) = r15;/*set valid slot bits in tag phase for slots 0,1,2 */
              r1 = dm(I4,2);       /* get indexed register address that is to be inspected */
              r2 = 0x8000;         /* set bit #15 for read request in command address word */
              r1 = r1 OR r2;       /* OR read request with the indirect register value */
              dm(tx_buf + COMMAND_ADDRESS_SLOT) = r1; /*send it out of command address timeslot*/
              idle;                /* wait for 2 audio frame to go by,latency in getting data */
              idle;
              r3 = dm(rx_buf + STATUS_ADDRESS_SLOT);
              dm(I5,1) = r3;
              r3 = dm(rx_buf + STATUS_DATA_SLOT);/* fetch requested indexed register data */
              dm(I5,1) = r3;                     /* store to results buffer */
ad1819_register_status: nop;                     /* wait until TDM frame is transmitted */
```

Explanation Of The AD1819A Codec Register Readback Loop :

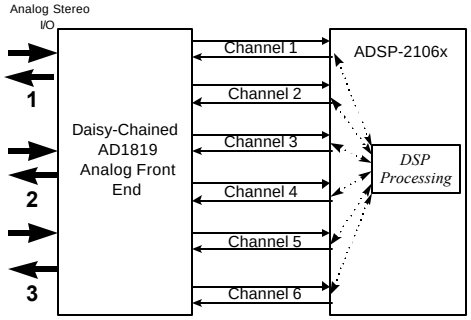
- The buffer pointers I4 and I5 is first set to point to the top of the codec register buffer and codec results buffer.
- Slot 0 (TAG Phase) is set up to enable the valid frame bit and slot 1 valid bit by writing a value of 0xC000 in DM(tx_buf + 0)
- The Loop Counter Register LCNTR is set to the number of registers to be read from the AD1819A. In this case 17 registers in all 3 codecs will be set to the same value.
- Memory writes from DM(tx_buf + 1) will set a read request for the codec register address specified in the *Init_Codec_Registers[ ]* buffer. Since we are modifying by two, we are only reading AD1819A register addresses from thins input buffer, OR'ing in the read request bit #15 (the MSB) and transmitting the read request of the address in the TX address timeslot.
- Two **IDLE** instructions are required to correctly readback the codec, since 1 TX audio frame is required to send the readback request, and 1 rx audio frame is then required to send the contents of the register requested in the next audio frame.
- Memory reads from DM(rx_buf + 1) will fetch the register read address
- Memory reads from DM(rx_buf + 2) will fetch the register read data for the codec address specified in the previous timeslot.
- The pointer I5 copies the register address and data in the *Codec_Init_Results[ ]* buffer for every read request. The resulting buffer contents alternates with the codec addresses on even DSP addresses, and codec data for the corresponding codec register address on odd DSP memory locations. This configuration is similar to that of the input *Init_Codec_Registers[ ]* buffer, so the user can than easily compare the codec programming buffer to the codec read results buffer.

6. Processing AD1819A Audio Samples via the SPORT Tx or Rx ISR

The 21065L's SPORT1 Interrupt Vectors and Interrupt Service Routines can be used to process incoming information from the up to 3 AD1819As through one serial port. As was described earlier in section 3.4, the information sent from the AD1819A is *DMA-Chained* (i.e., the SPORT receives the entire block of AD1819A frame data before a SPORT interrupt occurs, and the DMA settings are automatically reloaded to repeat the transfer of codec data) into the **rx_buf[ ]** buffer and an interrupt is generated when the buffer is filled with new data. Therefore, when a RX interrupt routine is being serviced the data from all active receive timeslots has been filled into the receive DMA buffer. When a TX interrupt routine is being services, the data from the DMA buffer has been transferred out to the serial port. Output left and right samples, codec commands, and valid tag information are filled into the transmit DMA buffer **tx_buf[ ]** for transmission out of SPORT. The programmer has the option of executing the DSP algorithm from either the transmit DMA interrupt or the receive DMA interrupt.

Figure 15 below shows a high level logical view of the audio streams that can be processed when interfacing up to 3 AD1819s to the SHARC DSP. With stereo ADCs and DACs on each codec, each SHARC serial port is capable of processing 6 input audio channels and send DSP output audio streams to 6 output channels. This type of multiple codec configuration can find applications in low-cost audio designs, such as implementation of a 6 x 2 channel digital mixing console, or provide a low-cost solution for running surround algorithms requiring 6 channels for audio playback. With the use or both SPORT0 and SPORT1, the ADSP-21065L can interface to up to 6 AD1819s, resulting in an audio system with 12 audio input and output channels.

Figure 15. High-Level View Of Three-AD1819 / SHARC Audio System



Using a DSP SPORT interrupt routine, the processor can detect if it has valid data from the codecs. The DSP's interrupt routine can then send audio data from a given input channel to that channels processing algorithm and places results back to the output channel stream (in the SPORT DMA buffer) for conversion by that AD1819A DAC. In this section we will investigate methods for processing data from either the SHARC's SPORTx receive interrupt vector or transmit interrupt vector.

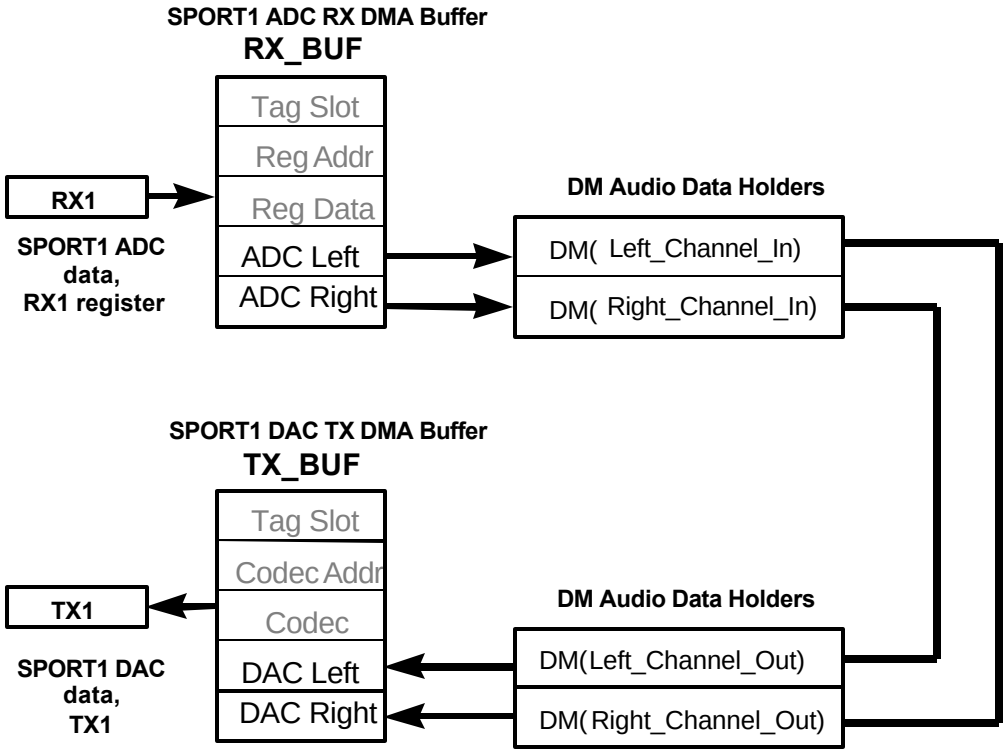
With AD1819A variable sample rate capabilities, it is possible to sample signals at one rate and playback the samples at another sampling rate. The example codec driver in Appendix A shows how the DSP can test to see if valid data is coming the current frame. For sampling rates less than 48 kHz, there will not always be valid data, so the DSP's SPORT audio processing ISR must allow for early return from interrupt with no processing if no valid data is detected in the current frame. For example, running at a 8 kHz sampling rate would result in valid data coming in once every 6 frames (48KHz Frame Rate / 8 kHz sampling rate = 6), so calls to DSP processing routines would be made every 1 out of 6 SPORT ISR calls.

For variable sample rate applications where there is not always a valid sample per audio frame, slot0 rx and tx valid TAG slot bits and left/right channel data slot synchronization becomes more crucial and cannot be ignored by the programmer, due to SPORT MCM & DMA timing relationships. For instance, the DSP TX interrupt routine may try to poll the L/R ADC valid bits in the rx buffer while the valid left and right data has not yet been DMA'ed into the receive buffer. Another problem occurs when processing audio in the SPORT's RX interrupt with only 5 slots/DMA words enabled. The DSP may try to set new tx valid slot tag bits in the transmit DMA buffer for transmission, but the slot 0 data has already been DMA'ed to the SPORT TX registers. So the tag slot information, instead of getting shifted out in the next audio frame, will remain in the tx DMA buffer and will not be transmitted until the following frame. Since it will have been too late to write tx tag data to the to the transmit DMA buffer because of these TDM, DMA and interrupt timing differences, the AD1819A will miss converting processed samples, resulting in minor or severe audible distortion. However, there are a few tricks we will discuss in this section that will enable the programmer to successfully write a variable sample rate codec driver. After extended testing and experimenting in search of the

most efficient driver implementations, we found using the SPORT TX interrupts to process data, when setting the codec to a sample rate less than the default 48 kHz., resulted in the lowest the DSP's IOP bus bandwidth utilization. However, rx-based processing with 16-word TX DMA buffers and 16 active TX TDM slots removes the restriction of having to read and write valid tag information early in the SPORT receive interrupt service routine.

An example AD1819A audio service routine data flow sequence (this is the procedure actually used in our reference ADSP-21065L codec drivers) for processing audio data in the SPORT1 transmit or receive ISR is shown below in Figure 16. This diagram shows the flow of audio samples from the serial port registers, to the DMA buffers, and from there, the audio samples are copied into temporary memory locations for a simple loopback, or to be used as inputs for the audio processing routines. The output data is then copied from the output queue into the transmit DMA buffer, where it is then transferred to the SPORT transmit data register for shifting out of the serial port. Again, the codec processing instructions can be serviced and executed from either the transmit or receive interrupt vector. If we assume the ADCs/DACs run at the 48-kHz default sampling rate, there is never a concern for TAG bit, ADC valid bit, and DAC request bit alignment. If all codecs are set to run at 48 kHz, then the DSP SPORT1 receive ISR can skip testing of audio data since valid bits in the TAG for ADC data should always be 1's.

Figure 16. Example 21065L EZ-LAB AD1819 Software Driver



Example SPORT1 RX or TX Interrupt Service Routine Workflow

*1) Initially clear transmit DMA buffer slots in **tx_buf** (stuff all slots with 0's if no data is sent, to conform to AC'97 spec)*

2) Check the AD1819 Stereo ADCs for valid data from Tag Phase in Slot 0. Also, instead of polling the ADC Valid Bits, the DAC request bits in address 0x74 can be checked to see if the DACs are requesting data (This only is applicable for processing data at sample rates less than 48 kHz)

3) Set the TX Tag Phase Bits in TX slot 0 to Notify AD1819 DACs that valid data is being transmitted in the current frame, depending on if either the ADCs containing data, or the DACs requesting data.

*4) Based on Step 2's results, get valid data from AD1819a via the SPORT receive DMA buffer **rx_buf***

5) Save Codec Data to Registers or Memory for DSP Processing

6) Run Desired DSP Algorithm

7) Transmit DSP Algorithmic Results to AD1819A DACs via the SPORT DMA buffer tx_buf.

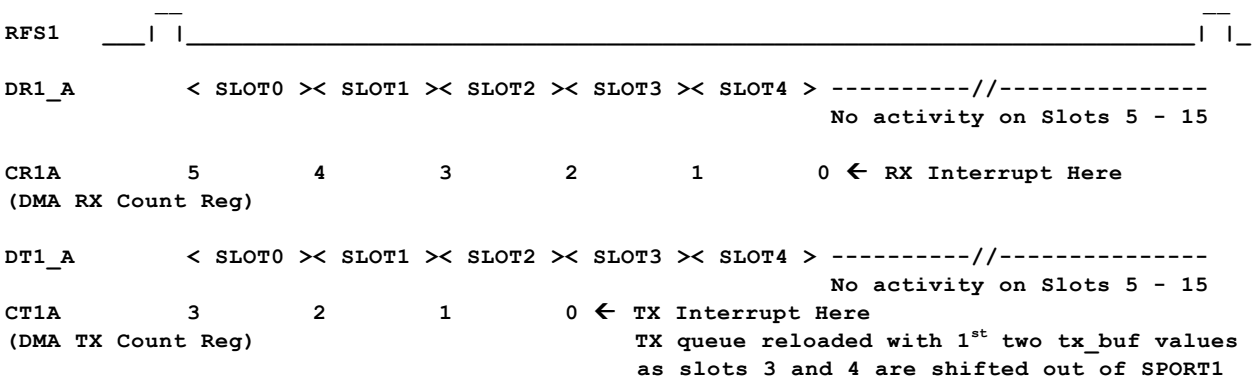
6.1 Important 21065L EZ-LAB SPORT1 DMA & AD1819A Multichannel Timing Notes

For a single codec system (such as the 21065L EZ-LAB AD1819A), the DSP programmer needs to be careful in the choice of implementing a custom driver. Ideally, we wish to implement the DMA buffers with the minimum amount of memory required with least possible number of TDM channels enabled, as we get a memory savings and a reduction in IOP transfer bandwidth utilization. For a single AD1819A codec system, this would require a TX and RX buffer of 5 words. In some cases, the programmer may discover that a driver that runs successfully at 48 kHz no longer runs correctly at slower sample rates. In other instances, a driver that may run clean (no distortion) at 16 kHz, an integer divisor of 48 kHz, will sound distorted at fractional sample rates of 48 kHz such a 8.124 kHz. This is because at fractional sample rate ratios of 48 kHz, the ADC valid bits are being set in unpredictable patterns compared the AD1819As DAC request bits, which may be set in alternative bit patterns. In this case the DSP algorithm result may accidentally be transmitted to the AD1819 twice or skip and output sample because of data overruns or under-runs. In this section we will DMA timing and various methods for processing data for successful DSP/Codec driver implementations. (The topics discussed here apply for multi-codec systems with additional DMA buffer words and slots enabled, but we will not discuss two or three codec system timing issues in detail, since all timing issues described here apply for multiple AD1819As interfaced to the SHARC SPORT)

For SPORT TX and RX DMA chaining in TDM mode, the DMA interrupts are always at least two timeslots apart (See Figure 17 below). This is because after the Transmit TCB is downloaded from memory and the reactivated SPORT transmit DMA channel prepares for data transmission, the DMA controller initially places the first two words from the SPORT transmit DMA buffer into the two SPORT1 TX buffer FIFO registers. This automatically decrements the Transmit DMA Count Register by two. After the assertion of the TX chained-DMA interrupt, we would need to wait until the active RX chained-DMA transfer brings in data for channels 3 and 4 for the current frame in which the rx ADC valid bits and DAC request bits are set, because the Receive DMA Count Register is always delayed behind the Transmit DMA Count Register by a value of two.

So, before timeslot 0 even begins transmit/receive activity, the **RX DMA Count = 5** while the **TX DMA Count = 3** (assuming we have declared 5-word TX and RX DMA buffers with 5 active rx and tx timeslots enabled on the serial port). The transmit interrupt occurs when the TX DMA Count = 0, and this interrupt request occurs on the second DSP clock cycle immediately after the LSB of timeslot 2 is received. While this transmit interrupt is generated, the transmit data for the left channel DAC timeslot is currently shifting out of the SPORT's TX-shift register in slot 2, while the AD1819 right channel TX DAC data for timeslot 4 is in the TX1A register queue, waiting to be transmitted after timeslot 2 data is finished shifting out of the SPORT. By the time both the transmit and receive interrupts are latched in the current frame (after timeslot 4), the rx and tx TCBs will be reloaded, but no DMA internal memory transfers will occur until the next frame sync, which would occur 11 time-slots later (with the exception of the 1st two words in the TX buffer, which reload into the TX SPORT shift register and data buffer TX1A as the previous channel 3 and 4 data shift out of the TX-shift register). After each reloading of the Transmit DMA TCB parameters, the first two values from the TX DMA buffer are automatically reloaded into the tx-shift register and the TX data buffer after the previous timeslots 3 & 4 data are shifted out of the SPORT TX-Shift register, but remain in the 'tx-queue' until the DSP generates the next 48 kHz frame sync within 11 16-bit timeslots.

Figure 17. AD1819/SPORT Timeslot, DMA Count and RX & TX Interrupt Timing Relationships



(1 / 12.288 MHz SCLK) x (16-bits/timeslot) x (2 timeslots) = 2.604 microseconds

1 / 60 MHz Instruction Execution = 16.667 nanoseconds per instruction

2.604 microseconds / 16.667 nanoseconds = 156.25 = 157 DSP cycles

6.2 Multichannel, DMA and ISR Methods of Implementation For Processing Data < 48 kHz

Now that we have examined in section 6.1 the relative timing difference in SPORT TX and RX interrupts between the transmit and receive channels, we will investigate various implementation methods to ensure proper data and tag alignment for processing data from the AD1819A. Note that these methods apply to the single codec case and they may become necessary to follow when running at slower sample rates. For multiple codecs, these recommendations still apply, but additional DMA buffer words and the number of active timeslots enabled in the TDM frame would need to be added (2 extra timeslots and DMA words for a dual codec system, and 4 extra timeslots and DMA words for a triple codec system).

1. Insert a Delay Loop
Use 5-word TX and RX DMA buffers with channels 0 - 4 active
Audio processing via the SPORT Transmit Interrupt

One method for guaranteeing new left/right rx data in slots 3 and 4 will be available when entering the SPORT TX interrupt and detecting corresponding ADC tag bits to be valid, is to implement a delay loop to simply wait for the data. This method is only applicable if we have declared a SPORT1 TX DMA buffer size of 5 along with 5 active TDM channels enabled in the SPORT TX multichannel enable register, equivalent in size to the SPORT1 RX DMA buffer and enabled rx channels. Immediately before fetching the left and right ADC data a delay loop can be added to wait for the data. This can be coded as follows:

```
                        LCNTR=126, Do Delay_Getting_Data UNTIL LCE;
    Delay_Getting_Data:     NOP;
```

This delay loop is only required if the user needs to run at sample rates < 48 kHz with a 48 kHz frame rate and when processing data from the TX interrupt while only declaring a TX DMA buffer size of 5 along with 5 TX TDM channels enabled.

It is estimated that while waiting for the left channel in slot 3, it takes 79 DSP Cycles at 60 MIPs. While waiting for the right channel data in slot 4, it takes approximately 157 DSP cycles.

Because of the interrupt latency and previous instructions to this point, it was found using the 21065L EZ-LAB that a Loop counter value of 126 guarantees enough time for the right channel data to be received.

This approach is probably not acceptable for most designs, but has been found to work for smaller audio applications. This is definitely true if the programmer needs to include other background processing or interrupt routines that need to be executed during this time. Thus this method is not the preferable method because of the loss in MIPs bandwidth (about 12% loss in bandwidth utilization). The programmer can should instead try steps 2 and 3. The loss in bandwidth is estimated as follows:

(60 MIPs / 48 kHz AC-97 Audio Frame Rate) = 1250 DSP cycles to process data
DSP cycles - 150 = 1100 DSP cycles, or about 12% loss in available MIPs

Recommended DSP Driver Settings:

```
.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* receive tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 5, 1, 0;      /* transmit tcb */

.var rx_buf[5];               /* receive buffer */

.var tx_buf[5] =              /* transmit buffer */
                  0xE000,     /* set valid bits for slot 0, 1, and 2 */
                  0x7400,     /* serial configuration register address */
                  0xFF80,     /* Slot-16 mode */
                  0x0000,     /* stuff other slots with zeros for now */
                  0x0000;
---

      /* sport1 receive and transmit multichannel word enable registers */
```

```
      R0 = 0x0000001F;        /* enable transmit and receive channels 0-4 */
      dm(MRCS1) = R0;
      dm(MTCS1) = R0;
---
      bit set imask SPT1I;    /* enable sport1 xmit  */
```

2. Use a TX DMA buffer = 7 words in length with TX slots 0-6 active
RX DMA buffer = 5 words in length with 5 active slots - 0-4
Audio Processing via the SPORT TX Interrupt

An efficient implementation is to extend the TX DMA buffer size to 7 in length and enable 7 active TX TDM channels. This will guarantee that by the time the DSP generates a TX DMA interrupt, the RX left and right data for the current frame have been received in time before reading the samples. Slots 5 and 6 are dummy slots and never used. However, the advantage to this implementation is that *the DSP core is not held up waiting for the RX left and right channels to be DMA'ed into RX_BUF*. (In order to implement this method with the 21065L EZ-LAB RS232 Debugger, the programmer is required to run a SPORT1 register clear routine to reset SPORT1 MCM and DMA activity… refer to Appendix A for an example SPORT1 register-clear routine)

Recommended DSP Driver Settings:

```
.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* receive tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 7, 1, 0;      /* transmit tcb, TX DMA count = 7  */

.var rx_buf[5];               /* receive buffer */

.var tx_buf[7] =              /* transmit buffer */
                  0xE000,     /* set valid bits for slot 0, 1, and 2 */
                  0x7400,     /* serial configuration register address */
                  0xFF80,     /* Slot-16 mode */
                  0x0000,     /* stuff other slots with zeros for now */
                  0x0000,
                  0x0000,     /* slots 5 and 6 are dummy slots */
                  0x0000;
---
      /* sport1 receive multichannel word enable registers */
      R0 = 0x0000001F;        /* enable receive channels 0-4 */
      dm(MRCS1) = R0;
      /* sport1 transmit multichannel word enable registers */
      R0 = 0x0000007F;        /* enable transmit channels 0-6 */
      dm(MTCS1) = R0;
---
      bit set imask SPT1I;    /* enable sport1 xmit  */
```

3. Use Both the SPORT RX and TX Interrupts for audio processing
RX ISR receives AD1819A Data and TAG/DAC reqister information
TX ISR transmits processed audio data to AD1819A DACs
Use 5-Word TX and RX DMA Buffers with rx/tx timeslots 0-4 enabled

Using this method, set the TX DMA buffer size and active MCM channels to 5, and then use both the SPORT1 tx and rx interrupts to send/receive AD1819a audio data. Refer to alternate AD1819 reference code for this solution. ADC Valid Bits or DAC Request Bits information can be passed from the SPORT receive ISR to the SPORT transmit ISR through a register or variable stored in memory.

Recommended DSP Driver Settings:

```
.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* receive tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 5, 1, 0;      /* transmit tcb */

.var rx_buf[5];               /* receive buffer */

.var tx_buf[5] =              /* transmit buffer */
                  0xE000,     /* set valid bits for slot 0, 1, and 2 */
                  0x7400,     /* serial configuration register address */
                  0xFF80,     /* Slot-16 mode */
                  0x0000,     /* stuff other slots with zeros for now */
                  0x0000;
---
      /* sport1 receive and transmit multichannel word enable registers */
      R0 = 0x0000001F;        /* enable transmit and receive channels 0-4 */
      dm(MRCS1) = R0;
      dm(MTCS1) = R0;
---
      bit set imask SPT1I | SPR1I;  /* enable both sport1 xmit and rcv  */
```

When using the TX ISR for audio processing or for transmitting output audio data to the AD1819A (using either methods 1, 2 or 3), the DSP programmer has approximately 75 DSP cycles (assuming 60 Mhz operation) upon entering the SPORT TX ISR to write new tag information to the DM(TX_BUF + 0), which is the TX TAG phase slot, in order that our left and right transmit data slots go out within the same audio frame as the TAG slot. If not done in time, the DSP would send data out in the current frame, but the tag bits would get sent in the following frame. The DSP would then risk the dropping of samples, and severe audible distortion results. The timing requirement for writing to the TX TAG location is estimated as follows:

(1 / 12.288 MHz SCLK) x (16-bits/timeslot) x (1 timeslots) = 1.302 microseconds

1 / 60 MHz Instruction Execution = 16.667 nanoseconds per instruction

microseconds / 16.667 nanoseconds = 156.25 = 78.125 DSP cycles

DSP cycle (RX interrupt request 1 CLKIN cycle after last bit of serial word is shifted out of SPORT)

4 cycles Interrupt Vector Latency (1 cycle sync and latch, 1 cycle recognition, 2 cycles to branch to int vector)

79 - 1 - 4 = 74 DSP Instruction Cycles to write to the TAG from the TX ISR

Therefore, when processing data at slower sample rates < 48 kHz, it is recommended to use the SPORT TX interrupt to process audio or transmit DAC data … but this is only required whenever you are using 5 or 7 word TX DMA buffer sizes. This reason for using the TX ISR instead of the RX ISR in this situation is as follows: If we were to use the RX ISR for processing, whenever the RX DMA interrupt occurs, the Tx TAG slot data in the top of the TX DMA buffer has already been loaded into the TX1A register waiting for the next frame sync. Thus it would be too late to try to write to the TAG DMA buffer offset because it has already been transferred by the DMA controller to the SPORT circuitry. This transfer would have occurred approximately 1 timeslot, or 16 BIT_CLK cycles prior to entering the RX Interrupt Service Routine.

For fixed 48 kHz sample rate applications, the RX interrupt can still be used to process codec data and run a selected DSP algorithm, since the RX and TX TAG bits and valid left/right data occur in every frame and will always be set. The programmer therefore would not be concerned with polling ADC valid bits or DAC request bits. A ' one-audio-frame delay' would result in the transmitting of the processed data, but in realtime the latency would be negligible. The DSP actually would be transmitting left or right data that would correspond to the TAG bits set in the previous audio frame and ISR, which does not adhere to the AC-97 specification. However, since the ADC valid bits and DAC valid bits are set, the programmer no longer is concerned with carefully setting them at the appropriate time since these bits will always be set.

4. Use the RX Interrupt for Audio Processing
5-word RX DMA Buffer & 5 RX slots (0 - 4) enabled
16-word TX DMA Buffer & 16 TX Slots (0 - 15) enabled

This method will allow the DSP programmer to use the RX interrupt for audio processing, without having to install a TX ISR jump or call routine at the SPORT TX interrupt vector after programming the AD1819A. Another benefit is that by processing the data from the RX interrupt is that we guarantee that data has left and right TAG information and left and right data has been received in a newly filled buffer, after the generation of the RX interrupt after the 5th slot (SLOT 4), while at the same time the TX interrupt will continue to send dummy 'zero's in timeslots 5 - 15. This will ensure that there is plenty of time to write to the TX TAG slot location DM(TX_BUF + 0) at any time in the DSP's RX ISR audio processing routine, as long as it is done before the completion of the transmission of SLOT14 TX data. After SLOT14 is transmitted, the ADSP-21065L will DMA the new TX TAG data into the SPORT's TX1A register, while SLOT 15 begins to be shifted out of the SPORT T-shift register for SLOT15 transmission.

With this approach the DSP programmer can safely wait for a duration of 10 x 16-bit timeslots in the RX ISR before writing a new value to DM(TX_BUF + 0), which will be transmitted in the following frame.

In terms of DSP cycles, this equates to:

SLOT 15 (TX TAG DMA'ed into TX1A register)

SLOT5 (RX DMA Count = 0)

1 DSP cycle (RX interrupt request 1 CLKIN cycle after last bit of serial word is shifted into SPORT r-shift register)

4 cycles Interrupt Vector Latency (1 cycle sync and latch, 1 cycle recognition, 2 cycles to branch to int vector)

SLOT 15 - SLOT5 = 10 x 16-bit slots = 160 SCLK cycles = 171.25 DSP Cycles

172 - 1 - 4 = 777 DSP Instruction Cycles

Therefore, the programmer has 778 DSP Instruction Cycles (assuming 60 MHz operation) within the 21065L SPORT RX Interrupt Service Routine for which the DSP is required to write the new transmit TAG slot information into DM(TX_BUF + 0) so that it correctly gets transmitted in the next audio frame. The advantage to this method is that the programmer is not restricted to having to write to the TX Tag location in the beginning of the ISR. The drawback is that DSP requires more memory for the TX DMA buffer, and DMA transfer overhead is increased from 5 or 7 transfers to 16 on the 21065L's IOP bus.

Recommended DSP Driver Settings:

```
.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* receive tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 16, 1, 0;     /* transmit tcb, TX DMA count = 16 */

.var rx_buf[5];                /* receive buffer */

.var tx_buf[16] =              /* transmit buffer */
                  0xE000,      /* set valid bits for slot 0, 1, and 2 */
                  0x7400,      /* serial configuration register address */
                  0xFF80,      /* Slot-16 mode */
                  0x0000,      /* stuff other slots with zeros for now */
                  0x0000,
                  0x0000,
                  0x0000,
                  0x0000,
                  0x0000,
                  0x0000,
                  0x0000,
                  0x0000,
                  0x0000,
                  0x0000,
                  0x0000,
```

```
                  0x0000,
                  0x0000;
---

      /* sport1 receive multichannel word enable registers */
      R0 = 0x0000001F;        /* enable receive channels 0-4 */
      dm(MRCS1) = R0;
      /* sport1 transmit multichannel word enable registers */
      R0 = 0x0000FFFF;        /* enable transmit channels 0-15 */
      dm(MTCS1) = R0;
---
      bit set imask SPR1I;    /* enable sport1 rcv  */
```

6.3 Using The *'Rx ADC Valid Bits'* Method Or The *'DAC Request Bits'* Method To Transmit Processed Audio Data To The AD1819A DACs (For Variable Sample Rates < 48 kHz)

When processing the audio data in the SPORT's transmit or receive interrupt service routine, the programmer has the option to either poll the ADC valid bits of the incoming data or the DAC request bits (in the Serial Configuration address 0x74) to determine when to transmit processed audio data to the AD1819A DACs in the next audio frame. To prepare DAC data to transmit in the next audio frame, the DSP's SPORT ISR instructions should simply include DM data transfers the appropriate locations in the SPORT transmit DMA buffer, which in turn is transferred out of the serial port on the next TDM frame sync assertion. The DSP's SPORT interrupt routine's codec-specific instructions either would poll the rx-ADC valid bits in the TAG slot (slot 0), or poll the DAC request bits in either Slot1 or Slot2. If either the ADC valid bits are set or DAC requests are made, the DSP then executes instructions to ensure that the tx TAG bits (slot 0) for the left and right DAC channels are set, since it will place processed data in the left channel slot (slot 3) and the right channel slot (slot4). We will look at both methods and offer advantages and disadvantages to using either method.

6.3.1 *'ADC Valid Bits'* Method

Polling the ADC Valid Bits in the Tag Phase (Slot 0) upon entering the SPORT/codec interrupt service routine will tell the DSP if it needs to fill the TX DMA buffer slots 3 and 4 with valid data for transmission in the next audio frame. Usually, we will first poll these bits to determine if we have to save and process our left/right audio data in slots 3 and 4. Once we save our new sample for processing, we will only transmit the ADC valid bit's corresponding DAC channel data if we have received new ADC samples. This method is more of a pipelined FIFO approach, in which we always will transmit the newly processed sample to the DACs in the next audio frame every time we get the new ADC sample and process it. Using the 'ADC valid bits' method usually yields a clean ADC-DSP-DAC loopback path at fractional variable sample rates < 48 kHz.

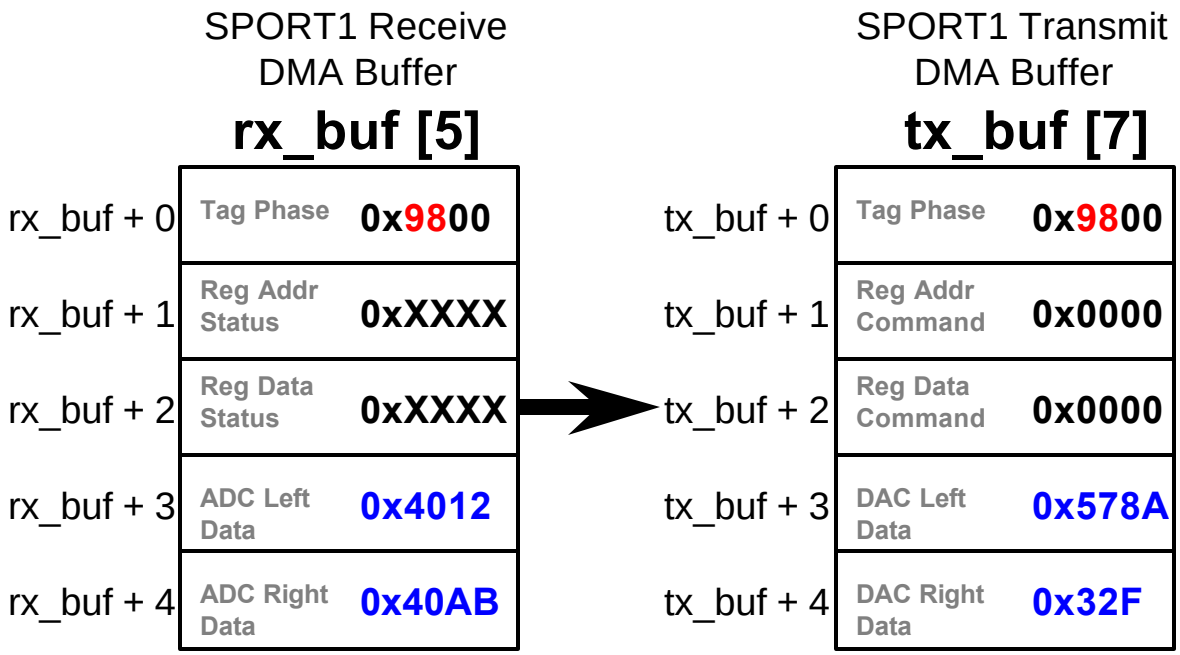
Figure 18 on the following page visually shows the 21065L SPORT transmit and receive DMA buffers at the point when the DSP vectors off to the SPORT interrupt to process data. Typically, you would see this sort of display in the 65L RS232 VDSP monitor debugger or JTAG VDSP EZ-ICE's *'two-column memory window'*. Inspecting the contents of the SPORT receive DMA buffer 'rx_buf' will give an indication if we have new ADC data. If these valid bits are set, then we ensure that corresponding DAC channel data will be available in the SPORT transmit buffer for transmission in the following audio frame.

The newly-filled SPORT receive DMA buffer contains new data from previous audio frame. We see that the ADC valid bits for slots 4 and 5 in the slot 0 Tag (or DM(rx_buf + 0)) are set as well as the Codec Ready bit D15, and valid data exists in slots 4, DM(rx_buf +3), and 5, DM(rx_buf + 4). Slots 1 and 2 are *'Don't Care'* conditions and ignored in this case. It will usually contain the status of the last codec read, or if the DRQEN bit is set, these slots will display the contents of address 0x74, the serial configuration register.

When the DSP ISR detects that there is valid data, it then calls the user's processing routine, or sets a flag to notify the background task that there is new data to process. Thus, for slower sample rates less than 48 kHz, we will only execute our DSP routine when the DSP detects valid data, and as a result, the DSP will only transmit DAC data when ADC data for that channel was valid for the previous frame and current SPORT ISR. Now, in examining the contents of the Transmit DMA buffer in the figure below, we see that new data is copied into the buffer and is transmitted on the next frame sync assertion as a result of

detecting valid data in the previous frame. Since there is valid ADC data in rx_buf[ ], the DSP ensures that the DAC valid Tag bits for slots 3 and 4, as well as the 'Valid Frame' bit D15, are set in DM(tx_buf +0). The interrupt routine also places any processed audio samples in Slots 3 and 4 for the Left and Right DAC channels. Unused slots are filled with zeros.

Figure 18. *'ADC Valid Bits'* Method For Transmitting DAC Data In The Next Audio Frame



We will now examine some example 21065L Assembly Language Instructions incorporated in our 21065L SPORT Interrupt Service Routine (listed in Appendix A), that show how to detect ADC valid bits and then set up the DAC slot tag bits for the transmission of DAC data in the next audio frame if the ADC valid bits are set.

1) These instructions check if we have new ADC data from the TAG slot 0 by reading from DM(rx_buf + 0). Masking out that value with 0x1800 tests the bit positions corresponding to slots 3 and 4:

```
check_ADCs_for_valid_data:
        r0 = dm(rx_buf + TAG_PHASE);     /* Get ADC valid bits from tag phase slot*/
        r1 = 0x1800;                     /* Inspect for valid L/R ADC data */
        r2 = r0 and r1;                  /* Mask other bits in tag */
       dm(ADC_valid_bits) = r2;
```

2) We then quickly set the tx TAG bits for slots 3 and 4. This will either set the tx TAG bits for the left and right channel to zero if no valid data, or we will set the DAC tag bits if the ADC valid bits were set, and thus we will be filling up the tx DMA buffer locations 3 and 4 with new left and right channel DAC data:

```
set_tx_slot_valid_bits:
        r1 = dm(tx_buf + TAG_PHASE);     /* set tx valid bits based on ADC valid bits info */
        r3 = r2 or r1;                   /* set left/right channel bits in tag, if required */
        dm(tx_buf + TAG_PHASE) = r3;     /* Write tag to tx-buf ASAP before it's shifted out
SPORT! */
```

3) Now that we set up the TX TAG bits, we save our current left and right channel data for processing. We will only save our data whenever we detect that we have valid data. This is done by using the SHARC's shifter 'Bit Test' instruction, then testing to see if the shifter result was zero. If it is zero, we have no valid data, so we move on. If the result was a '1', then we save our newly detected sample. This is done as follows:

```
check_AD1819_ADC_left:
       BTST r2 by M_Left_ADC;            /* Check Master left ADC valid bit */
```

```
       IF sz JUMP check_AD1819_ADC_right;/* If valid data then save ADC sample */
          r6 = dm(rx_buf + LEFT);        /* get Master 1819 left channel input sample */
          r6 = lshift r6 by 16;          /* shift up to MSBs to preserve sign in 1.31 format */
          dm(Left_Channel_In) = r6;      /* save to data holder for processing */

check_AD1819_ADC_right:
       BTST r2 by M_Right_ADC;           /* Check Master right ADC valid bit */
       If sz rti;                        /* If valid data then save ADC sample */
          r6 = dm(rx_buf + RIGHT);       /* get Master 1819 right channel input sample */
          r6 = lshift r6 by 16;          /* shift up to MSBs to preserve sign in 1.31 format */
          dm(Right_Channel_In) = r6;     /* save to data holder for processing */
```

4) We then call our DSP algorithm. This can be conditionally called only if we have detected new audio data:

```
user_applic:
       call DSP_Audio_Routine;
       /* ---- DSP processing is finished, now playback results to AD1819 ---- */
```

5) After processing our ADC data, we now re-test the ADC valid bits to determine if we needed to send our results in the next audio frame. If these bits are set to a '1', we copy our results to the left and right DAC channels to slots 3 and 4 in the SPORT1 transmit DMA buffer, where it will await transmission to the AD1819A DACs through the AC-link.

```
Playback_audio_data:
       /* Transmit Left and Right Valid Data every time there the ADCs have valid data */
       r2 = dm(ADC_valid_bits);

tx_AD1819_DAC_left:
       BTST r2 by M_Left_ADC;            /* Check to see if we need to send DAC right sample */
       IF sz JUMP tx_AD1819_DAC_right;   /* If valid data then transmit DAC sample */
       r15 = dm(Left_Channel_Out);/      /* get channel 1 output result */
       r15 = lshift r15 by -16;          /* put back in bits 0..15 for SPORT tx */
       dm(tx_buf + LEFT) = r15;          /* output right result to AD1819a Slot 3 */

tx_AD1819_DAC_right:
       BTST r2 by M_Right_ADC;           /* Check to see if we need to send DAC right sample */
       If sz jump tx_done;               /* If valid data then transmit DAC sample */
       r15 = dm(Right_Channel_Out);      /* get channel 2 output result */
       r15 = lshift r15 by -16;          /* put back in bits 0..15 for SPORT tx */
       dm(tx_buf + RIGHT) = r15;         /* output right result to AD1819a Slot 4 */
```

Thus, the *'ADC Valid Bits'* Method provides an easy and predictable way to transmit DAC data in the next audio frame. This method usually yields clean audio without any Tag/Data/DMA timing issues that can cause distortion for sample rate values less than 48 kHz. Ring buffers are not required for fractional sample rate ratios. There are limitations to using this method to process audio data. The ADC and DAC channels (LADC/LDAC and RADC/RDAC) can only run at the same sample rates. You cannot run both the ADCs and DACs at different rates unless the AD1819A's DAC request bits are polled. We will cover this method in detail in the following section.

6.3.2 *'DAC Request Bits'* Method

Polling the DAC Request Bits in the Serial Configuration Register (automatically in the Status Address and Status Data slots when the DRQEN bit is set) when we enter our SPORT interrupt service routine will tell the DSP if it needs to fill the TX DMA buffer slots 3 and 4 with valid data for transmission in the next audio frame. First, the ADC valid bits should be polled to determine if we have to save and process our left/right audio data in slots 3 and 4. Once the valid new samples are detected and saved for processing, we will only transmit DAC data if the DAC Request Bits were set in the AD1819A's Serial Configuration Register 0x74. This method is fairly pipelined at integer ratio sample rates of 48 kHz so that we never have any ADC/DAC sample overruns or underruns, while for fractional sample ratios the use of ring buffers to handle ADC/DAC sample rate jitter is recommended (see next section). The obvious benefit gained with the use of polling the DAC request bits, is that we are free to set left/right ADCs to different sample rates than the DAC, thus removing sample rate conversion routines on the DSP.

Figure 19. *'DAC Request Bits'* Method For Transmitting DAC Data In The Next Audio Frame

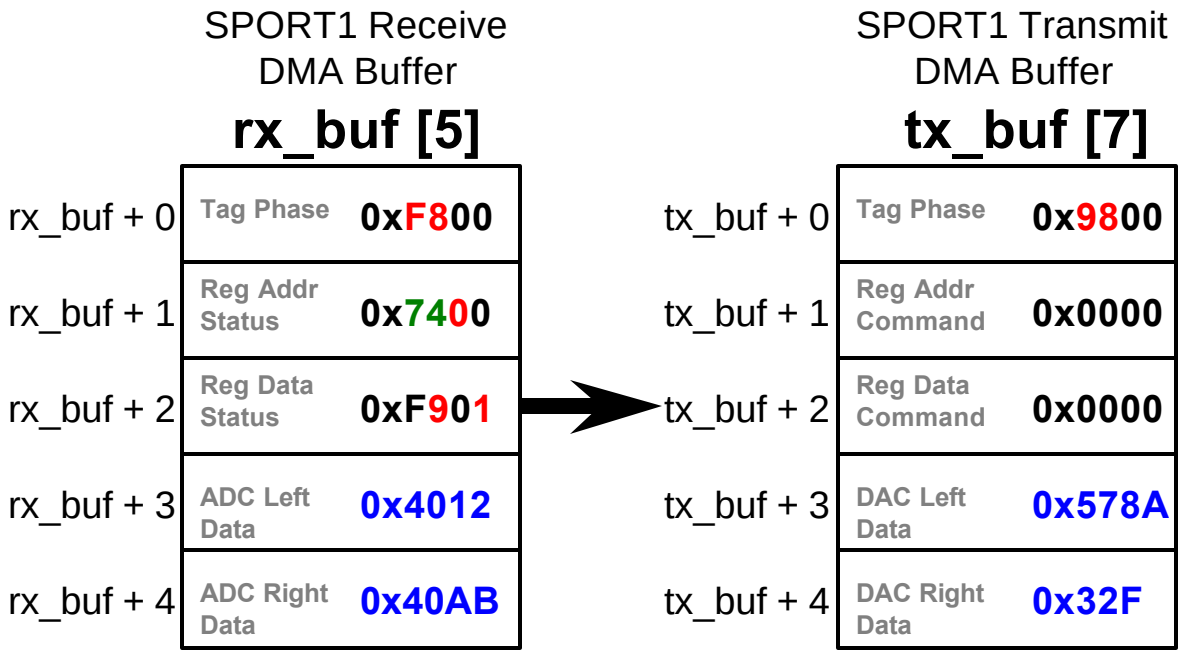


Figure 19 shown above visually shows the 21065L SPORT transmit and receive DMA buffers at the point when the DSP vectors off to the SPORT interrupt to process data. As stated before, you would see this sort of display in the 65L RS232 VDSP debugger or JTAG VDSP ICE's *'two-column memory window'*. Inspecting the contents of the SPORT receive DMA buffer 'rx_buf' will give an indication if we have new ADC data, and if the AD1819A DACs are requesting data via the DAC request bits. If these DAC request bits are set, then we ensure that corresponding DAC channel data will be available in the SPORT transmit buffer for transmission in the following audio frame.

The newly-filled SPORT receive DMA buffer contains new data from previous audio frame. We see that the ADC valid bits for slots 3 and 4 in the slot 0 Tag (or DM(rx_buf + 0)) are set as well as the Codec Ready bit D15, and valid data exists in slots 3, DM(rx_buf +3), and 4, DM(rx_buf + 4). Slots 1 and 2 contain information on the DAC request bits through the automatic display of the Serial Configuration Register (Codec address 0x74). This data is automatically displayed when the DRQEN bit is set in this register.

When the DSP detects that there is valid ADC data (or when it detects the DAC request bits are set), it can call the user's processing routine, or set a flag to notify the background task that there is new data to process. So the DSP programmer can process data when either the ADC valid bits or DAC request bits are set in the current audio frame. Thus, for slower sample rates less than 48 kHz, the DSP will only execute the DSP filter routine it detects valid data (or DAC requests), and as a result, it will only transmit DAC data when ADC data for that channel was valid for the current audio frame SPORT ISR. Now, in examining the contents of the Transmit DMA buffer in the figure above, we see that new data is copied into the buffer and is transmitted on the next frame sync assertion as a result of detecting valid data in the previous frame. Since there is new valid DAC requests for both the left and right channels in rx_buf[ ], the DSP will ensure that the DAC valid Tag bits for slots 3 and 4, as well as the 'Valid Frame' bit D15, are set in DM(tx_buf +0). The interrupt services routine also places any processed audio samples in Slots 3 and 4 for the Left and Right DAC channels. Unused slots are filled with zeros.

We will now examine some example 21065L Assembly Language Instructions incorporated in our SPORT Interrupt Service Routine (listed in Appendix A) that show how to detect active DAC Request Bits, and then set up the transmit DAC data tag slot bits for the transmission in the next audio frame:

1) These instructions are used to poll the "Active Low" L/R DAC Request bits via the Status Address Slot 1. We shift these up by 5 bits, invert the values, so we can then used them to set the tx TAG bits for the left and right DAC timeslots 3 & 4.

```
Check_DAC_request_bits:
       r1 = dm(rx_buf + STATUS_ADDRESS_SLOT);  /* Get ADC request bits from address slot */
       r2 = r1 and r0;                    /* Mask out the AD1819 Master DRRQ0 and DLRQ0 bits */
       r2 = r2 xor r0;                    /* Set active low DAC request bits to active hi */
       r2 = lshift r2 by 5;               /* shift up so output tag info is bits 12 and 11 */
```

2) We then copy the shifted DAC request information to notify the AD1819 if slots 3 and 4 will contain valid data. If the bits are set, then we will copy DAC data in the current ISR. This gets copied into the Tx Tag Phase Slot 0.

```
Set_TX_slot_valid_bits:
       r0 = 0x8000;                       /* Write tag to tx-buf ASAP before it's shifted out! */
       r2 = r2 or r0;                     /* set tx valid bits based on received DAC request info
*/
        dm(tx_buf + TAG_PHASE) = r2;      /* Set Valid Frame & Valid Slot bits in slot 0 tag phase
*/
```

3) We then poll the Rx Tag Slot 0 to see if we have new L/R ADC data and save the data if valid.

```
       R0 = dm(rx_buf + TAG_PHASE);   /* get tag information to inspect for valid L/R ADC data */
```

4) As we saw before with the 'ADC Valid Bits Method', if we detect valid data (bit value is a '1') we save our current left and/or right channel data for processing. The bit validity is tested with the SHARC's barrel shifter:

```
check_AD1819_ADC_left:
       BTST r0 by M_Left_ADC;             /* Check Master left ADC valid bit */
       IF sz JUMP check_AD1819_ADC_right;/* If valid data then save ADC sample */
          r6 = dm(rx_buf + LEFT);         /* get Master 1819 left channel input sample */
          r6 = lshift r6 by 16;           /* shift up to MSBs to preserve sign in 1.31 format */
          dm(Left_Channel_In) = r6;       /* save to data holder for processing */

check_AD1819_ADC_right:
       BTST r0 by M_Right_ADC;            /* Check Master right ADC valid bit */
       IF sz jump user_applic;            /* If valid data then save ADC sample */
          r6 = dm(rx_buf + RIGHT);        /* get Master 1819 right channel input sample */
          r6 = lshift r6 by 16;           /* shift up to MSBs to preserve sign in 1.31 format */
          dm(Right_Channel_In) = r6;      /* save to data holder for processing */
```

5) We then call our DSP algorithm. This can be conditionally called only if we have detected new audio data:

```
user_applic:
       call (pc, DSP_Audio_Routine);
       /* ---- DSP processing is finished, now playback results to AD1819 ---- */
```

6) After processing our ADC data, we now test the DAC Request bits to determine if we needed to send our results in the next audio frame. If these bits are set, we copy our results to the left and right DAC channels to slots 3 and 4 in the SPORT1 transmit DMA buffer, where it will await transmission to the AD1819A DACs through the AC-link.

```
Playback_audio_data:
       /* Transmit Left and Right Valid Data if Requested */
       r2=DAC_Req_Left;                   /* Check to see if Left DAC REQ? */
       r3=r1 and r2;                      /* DAC request is active low */
       if ne jump bypass_left;            /* if it is 1, it means we have no request, so move on*/
          r15 = dm(Left_Channel_Out);     /* get channel 1 output result */
          r15 = lshift r15 by -16;        /* put back in bits 0..15 for SPORT tx */
          dm(tx_buf + LEFT) = r15;        /* output right result to AD1819a Slot 3 */

bypass_left:
       r2=DAC_Req_Right;                  /* Check to see if Right DAC REQ? */
       r3=r1 and r2;                      /* DAC request is active low */
       if ne jump bypass_right;           /* if it is 1, it means we have no request, so move on*/
          r15 = dm(Right_Channel_Out);    /* get channel 2 output result */
          r15 = lshift r15 by -16;        /* put back in bits 0..15 for SPORT tx */
          dm(tx_buf + RIGHT) = r15;       /* output right result to AD1819a Slot 4 */
```

6.3.3 Single (Master) AD1819A ADC Valid / DAC Request Reference Charts

The Tables below can be used as a reference for inspecting ADC Valid Bits, DAC Request Bits and audio data during debugging of a Single (Master) AD1819A Codec System:

Note: for single codec systems, the DAC request bits are zero'ed out for both the Slave1 and Slave2 bit locations. The AD1819A Master Codec stuffs zeros in these bit locations. In the cases below (which were observed with the Target65L RS232 VDSP Debugger), Slot16 mode was enabled, and all RegMx bits are set, even though there is only one master. Status Address Slot DAC Request Bits are 'Active Low'. Status Data Slot DAC Request Bits are 'Active High'

Status Address Slot #1 for Single Codec System with L & R DAC requests, inactive '0' bits for slaves 1 & 2

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	1	0	1	0	0	**0** Dreq Slot3	**0** Dreq Slot4	X '0'	X '0'	X '0'	X '0'	0	0

Status Data Slot #2 for Single Codec System with Left and Right DAC requests, inactive bits for slaves 1 & 2

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
1 Slot16	X REGM2	X REGM1	1 REGM0	1 DRQEN	X '0'	X '0'	1 DLRQ0	X '0'	X '0'	X '0'	X '0'	X '0'	X '0'	X '0'	1 DRRQ0

Table 10. Comparison of ADC Valid and DAC Request Bits

'ADC Valid Bits' Method:

Current Audio Frame (Read SPORT rx buffer) **Next Audio Frame (fill SPORT tx buffer)**

ADC Valid Tag Bits	*DAC Request Bits*	*Left Channel Timeslot 3*	*Right Channel Timeslot 4* →	*Tx Tag Slot 0*	*Left DAC Timeslot 3*	*Right DAC Timeslot 4*
0x9800	**0xXXXX**	**Valid Data**	**Valid Data**	**0x9800**	**Valid Data**	**Valid Data**
0x8800	**0xXXXX**	**Valid Data**	**No Data**	**0x8800**	**Valid Data**	**Zero Fill**
0x9000	**0xXXXX**	**No Data**	**Valid Data**	**0x9000**	**Zero Fill**	**Valid Data**
0x8000	**0xXXXX**	**No Data**	**No Data**	**0x8000**	**Zero Fill**	**Zero Fill**

'DAC Request Bits' Method:

Current Audio Frame (Read SPORT rx buffer) **Next Audio Frame (fill SPORT tx buffer)**

Rx Valid Tag Bits	*DAC Request Bits (slots 1 & 2)*	*Left Channel Timeslot 3*	*Right Channel Timeslot 4* →	*Tx Tag Timeslot 0*	*Left DAC Timeslot 3*	*Right DAC Timeslot 4*
0xF800	**[1] 0x7400 [2] 0xF901**	**Valid Data**	**Valid Data**	**0x9800**	**Valid Data**	**Valid Data**
0xF800	**[1] 0x7440 [2] 0xF900**	**Valid Data**	**Valid Data**	**0x9000**	**Valid Data**	**Zero Fill**
0xF800	**[1] 0x7480 [2] 0xF801**	**Valid Data**	**Valid Data**	**0x8800**	**Zero Fill**	**Valid Data**
0xF800	**[1] 0x74c0 [2] 0xF800**	**Valid Data**	**Valid Data**	**0x8000**	**Zero Fill**	**Zero Fill**
0xF000	**[1] 0x7400 [2] 0xF901**	**Valid Data**	**No Data**	**0x9800**	**Valid Data**	**Valid Data**
0xF000	**[1] 0x7440 [2] 0xF900**	**Valid Data**	**No Data**	**0x9000**	**Valid Data**	**Zero Fill**
0xF000	**[1] 0x7480 [2] 0xF801**	**Valid Data**	**No Data**	**0x8800**	**Zero Fill**	**Valid Data**
0xF000	**[1] 0x74c0 [2] 0xF800**	**Valid Data**	**No Data**	**0x8000**	**Zero Fill**	**Zero Fill**
0xE800	**[1] 0x7400 [2] 0xF901**	**No Data**	**Valid Data**	**0x9800**	**Valid Data**	**Valid Data**
0xE800	**[1] 0x7440 [2] 0xF900**	**No Data**	**Valid Data**	**0x9000**	**Valid Data**	**Zero Fill**
0xE800	**[1] 0x7480 [2] 0xF801**	**No Data**	**Valid Data**	**0x8800**	**Zero Fill**	**Valid Data**
0xE800	**[1] 0x74c0 [2] 0xF800**	**No Data**	**Valid Data**	**0x8000**	**Zero Fill**	**Zero Fill**
0xE000	**[1] 0x7400 [2] 0xF901**	**No Data**	**No Data**	**0x9800**	**Valid Data**	**Valid Data**
0xE000	**[1] 0x7440 [2] 0xF900**	**No Data**	**No Data**	**0x9000**	**Valid Data**	**Zero Fill**
0xE000	**[1] 0x7480 [2] 0xF801**	**No Data**	**No Data**	**0x8800**	**Zero Fill**	**Valid Data**
0xE000	**[1] 0x74c0 [2] 0xF800**	**No Data**	**No Data**	**0x8000**	**Zero Fill**	**Zero Fill**

6.4 Processing Data At Fractional (Versus Integer) Variable Sample Rate Ratios Via The *'DAC Request Bits'* Method

(I would like to offer thanks to Al Clark, of Danville Signal Processing, for his suggestions in using ring buffers to process AD1819A data at fractional sample rate ratios)

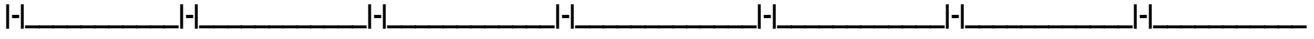
The above methods work well at slower sample rates that are integer ratio values of 48 kHz. Methods 1 - 4 described section 6.2, and in our source code examples in Appendix A, execute correctly for processing data set up for AD1819A sample rates of 16 kHz, 8 kHz, or 12 kHz. For example, when we program the AD1819A ADCs and DACs to run at a sample rate of 8 kHz, we expect to receive a sample once every 6 frames and we are required (or requested by an AD1819a DAC request bit) to transmit data once every 6 audio frames. For 16 kHz, we would expect to transmit or receive data once ever 3 frames, and at 12 kHz, once every 4 audio frames. Normally for these even spacing of samples, we would not ever risk any sample overruns or underruns because data will be evenly pipelined in and out of the DSP.

However, when running the AD1819A at fractional ratios of 48 kHz using the ADC valid bits to receive ADC data and the DAC request bits to transmit DAC data, an occasional sample repeat or sample miss can occur. This is because received ADC valid data and AD1819A DAC requests can be random relative to one another on the ADCs and DACs, for sample rates of, say, 23456 Hz, 8201 Hz, or 44100. When processing data in an interrupt service routine and polling the fetched ADC Valid Bits from the receive TAG slot, we only execute our DSP routine every time we get valid ADC data. After processing the current valid samples, we would normally place our results in the TX DMA buffer for the DACs to use when ready. The problem is, the DACs may not have been requesting data via the AD1819A DAC request bits in the next audio frame, so we risk occasionally drop a processed sample. Also, the ADC valid bits may not be set for a given frame, so no processing is done, but at the same time, we could get two consecutive 'DAC request bit' audio frames. This then would result in repeating back a DSP output sample twice, since the DSP algorithm was not processed in the current interrupt service routine. Keep in mind, this has nothing to do with the 'REPEAT VS ZERO FILL SAMPLE' bit that is in the in the AD1819A's Miscellaneous Control Bits Register if the DAC is starved. That functionality is for controlling the internal operation of the AD1819A DACs if the DSP did not send a requested sample in a given frame. Of course, if the input ADC and output DAC sample rates are different, we would then expect some ISR interpolation or decimation of samples to occur, in which we are repeating or skipping processed samples back to the AD1819A output DACs from the processing routine because the DACs are running at a different sample rate than the ADCs. Here, we are only concerned about the specific case where the ADCs sample rate is equivalent to the DACs, and we want to ensure *every new ADC sample is processed and all processed samples are sent back to the AD1819A*.

For example, if we were to look at a scenario (Figure 20) where ADC Valid Bits being set at non-integer ratios of 48 kHz by the AD1819A, while at the same time, DAC requests are made by the AD1819A DACs to keep DAC data filled at the same non-integer sample rate ratio, we may see something like this:

Figure 20. Non-Periodic ADC Valid & DAC Request Bit Patterns

SYNC Pulse



Frame N	Frame (N+1)	Frame (N+2)	Frame (N+3)	Frame (N+4)	Frame (N+5)	Frame (N+6)
ADC L/R Val.	ADC L/R Val.	No ADC val.	ADC L/R Val.	No ADC val.	No ADC val.	ADC L/R Val.
No DAC Req.	No DAC Req.	L/R DAC Req.	L/R DAC Req.	No DAC Req.	L/R DAC Req.	L/R DAC Req.
Left Ch. Data	Left Ch. Data	0x0000 L	Left Ch. Data	0x0000 L	0x0000 L	Left Ch. Data
Right Ch. Data	Right Ch. Data	0x0000 R	Right Ch. Data	0x0000 R	0x0000 R	Right Ch. Data

Note: 0x0000 L = No Data for left channel timeslot in current audio frame
0x0000 R = No Data for right channel timeslot in current audio frame
The AD1819A stuffs invalid slots with 'zeros'

Notice in Figure 20 above that in two consecutive audio frames, 2 L/R ADC valid bits may be set back-to-back, while 1 or 0 L/R DAC request will be made in the same two audio frames. Another possible scenario that can occur is that for two consecutive audio frames, 1 or 0 L/R ADC valid bits may be set while at the same time while 2 consecutive L/R DAC requests are made in the same two audio frames. It is exactly these two cases that can occur when sample rates for the ADCs and DACs are set to fractional ratios, such as 12345 Hz. This can cause 2 problems:

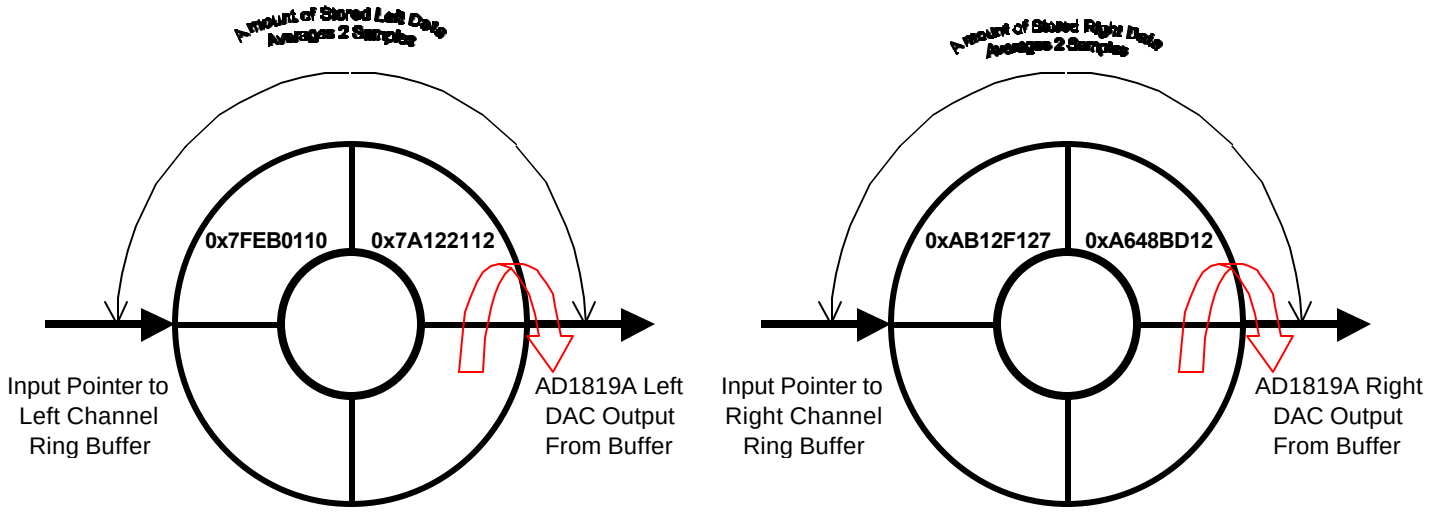
1) The DSP may process two consecutive ADC samples, and place the result in a holding register (in the case of our DSP driver, these are variables in memory) waiting for the AD1819A DAC to request it, but if the DAC does not make a request, then there is a possibility that that processed sample will be overwritten in the next audio processing interrupt routine.

2) The DSP may not receive ADC samples 1 or 2 ADC samples in a given frame, however the AD1819A DACs make two consecutive DAC requests for the processed audio data waiting in the DSP's holding register (or variable in memory). In this case there is a possibility that a processed sample will be transmitted to the DACs twice, before we are able to replace it with an updated processed sample.

The solution for this *'equal ADC/DAC fractional sample rate ratio case*' when using the *'DAC Request Bits'* method is to use a ring buffer. A ring buffer is a piece of allocated memory that is designed so that it's addressing of data periodically is allowed to overflow (or underflow) to a certain predetermined number of locations. Ring buffers are often used to prevent sampling jitter in sampling clocks, which can affect the quality of an audio signal. With the insertion of a ring buffer, the unpredictability of the valid or requested data in a given audio frame between the ADCs and DACs are no longer affected. The memory buffer increases the DSP's capturing of ADC data because sample frequency variations of the ADCs and DACs are absorbed by the buffer. Valid AD1919A Left/Right DATA and AD1819A DAC request variations at fractional sample rates of 48 kHz can cause the input and output data rate to vary independent in audio frames.

To ensure we have a smooth flow of data running at fractional ratios of 48 kHz, the programmer can implement a small ring buffer (Figure 21) for the left and right channel to prevent an occasional sample repeat or miss when running at fractional rates. A recommended scheme that has been found to work is to create one or two (depending on if we are performing mono or stereo processing) small *4-word (or up to 8-word) ring buffers* in the 'FETCH ADC DATA' and 'SEND DAC DATA' sections of the AD1819A processing routine (see figure below). The small circular ring buffers are initialized such that the output pointer offset the input pointer by 2 locations in memory. This would at least guarantee, for applications where the AD1819A ADCs are set to the same fractional sample rate as the DACs, that the input pointer would never pass the output pointer, but may vary 1, 2 or 3 samples ahead from the output pointer. The *effective delay* will always be two samples, which means we would always be between 1 to 3 '48-kHz' audio frames behind in getting an input sample into the DSP, and sending the result back out. This delay, of course, in real-time processing is too negligible to affect the listener for real-time audio applications.

Figure 21. AD1819A DAC Left and Right Channel Ring Buffers



Below is an example ring buffer implementation for the left and right channels. Note that this code is not optimized. The intermediate pointer states for the input and output are restored and saved using the same DAG index register I0, so that we are not using 4 separate DAG index registers on the DSP to implement these ring buffers input and output taps. With some additional overhead we only need to use one index register. The programmer can use a dedicated index register for each input and output pointer if there are enough available for their application and then remove all memory pointer save and restore instructions. Appendix A also includes ring buffer source code for use with the 'DAC Request Bits' Method.

```
.segment/dm dm_data;
.var Lchan_ring_buff[4] = 0, 0, 0, 0;
.var Rchan_ring_buff[4] = 0, 0, 0, 0;
.var L_input_ptr;
.var L_DAC_output_ptr;                          /* temporary storage of Index register */
.var R_input_ptr;                               /* this saves us from using 2 DAGs */
.var R_DAC_output_ptr;
.endseg;
-------------------------------------------------------------------------------------------
.segment/pm pm_code;
/* initialize the ring buffer input and output pointers */

         B0 = Lchan_ring_buff;
         DM(L_input_ptr) = I0;
         I0 = Lchan_ring_buff + 2;              /* start output ptr in middle of the buffer */
         DM(L_DAC_output_ptr) = I0;

         B0 = Rchan_ring_buff;
         DM(R_input_ptr) = I0;
         I0 = Rchan_ring_buff + 2;              /* start output ptr in middle of the buffer */
         DM(R_DAC_output_ptr) = I0;
         L0 = 4;                                /* both ring buffers are 4 words deep */
-------------------------------------------------------------------------------------------
/* these instruction can be added in the AD1819A Interrupt Service Routines where codec data is received, processed and transmitted */
         L0 = 4;                                /* input and output ring buffers are 4 words deep */
left_ring_buffer_input:
         R1 = DM(rx_buf + LEFT_CHANNEL);
         B0 = Lchan_ring_buff;
         I0 = DM(L_input_ptr);
         DM(I0,M1) = R1;
         DM(L_input_ptr) = I0;

right_ring_buffer_input:
         R1 = DM(rx_buf + RIGHT_CHANNEL);
         B0 = Rchan_ring_buff;
         I0 = DM(R_input_ptr);
         DM(I0,M1) = R1;
         DM(R_input_ptr) = I0;

left_ring_buffer_output:
         B0 = Lchan_ring_buff;
         I0 = DM(L_DAC_output_ptr);
         R1 = DM(I0,M1);
         DM(L_DAC_output_ptr) = I0;
         DM(tx_buf + LEFT_CHANNEL) = R0

right_ring_buffer_output:
         I0 = DM(R_DAC_output_ptr);
         R0 = DM(I0,M1);
         DM(R_DAC_output_ptr) = I0;
         DM(tx_buf + RIGHT_CHANNEL) = R0;
.endseg;
```

NOTE:

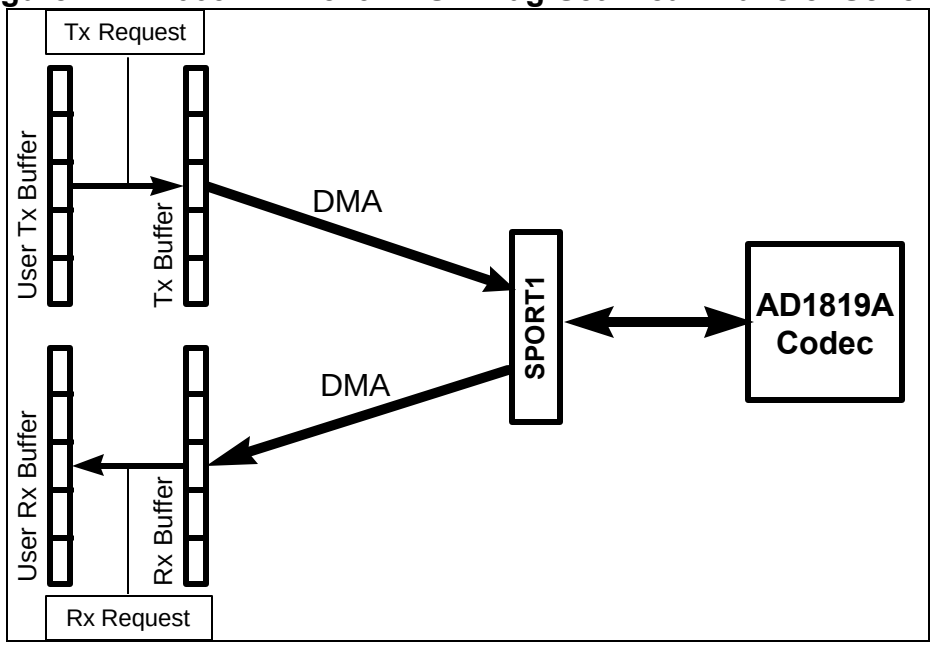
Based on our ADI DSP Applications Group's 'loopback' listening tests for low distortion (i.e. clean audio) at fractional sample rate ratios using the 'ADC Valid Bits for DAC transmission' Method, it was determined that the use of Ring Buffers is not really required. With this method of DAC transmission, a Left/Right DAC sample is only transmitted in the next audio frame whenever we receive a new Left/Right ADC sample. This appears to properly pipeline input samples from the SPORT into the 21065L's DSP algorithm and back out of the SPORT without dropping (or rarely dropping) samples, because the AD1819A's Left and Right DAC 'conversion hold' register will never be overwritten with a new value received through the AD1819A's serial interface while waiting to convert the previous DAC sample back to an analog signal. With this alternate method, the 'valid' ADC L/R sample timeslots received in the previous frame with always ensure that the DSP will transmit 'valid' L/R DAC timeslots to the AD1819A in the next audio frame.

Thus, the 'ADC Valid Bits' Method does not require the use of ring buffers when the ADC sample rate is equivalent to the DAC sample rate for both channels. When the ADC and DAC sample rates are the same for a given channel, this results in the ADC valid bit patterns being identical to the DAC Tag L/R valid bits which are set in the following frame. Since these Tag Bits are 'in phase', we never really run into a situation where a processed sample is skipped or repeated in the D/A conversion.

6.5 Using RX ADC Valid Flag And TX DAC Valid Flag Variables For Processing Audio Data At Variable Sample Rates

In certain applications, the user may want to process codec data elsewhere. For example, in C-based applications, the C-runtime DSP routines may be placed in a main program 'while' loop waiting for SPORT interrupts. The codec interrupt service routine's responsibility would be to receive and transmit codec data, while the processing of the data is done elsewhere. For slower sample rates, the DSP processing routine would need to know which SPORT interrupt actually received valid data. The DSP processing routine would also be responsible for notifying the codec transmit routine that it has valid processed data that can be transmitted back to the DACs. In order to implement such a scheme, the user can define variables that can be used as transmit and receive request flags that are set when data is ready, and cleared after the data has been transferred. For example, the 21065L demo examples (Figure 22) use a dual buffer scheme, which allows the user to copy data into a temporary buffer when the Rx Request variable has been set by the codec receive interrupt routine, while the DMA buffers are currently being filled, the user processed data from alternate background buffers. After audio data is processed, the information is copied to the transmit user buffer, and the Tx request variable is set. The codec processing routine detects that valid data is transferred into the user output buffer, and copied the data into the SPORT transmit DMA buffer for transmission to the AD1819. It is the responsibility of the DSP processing routine to clear any set rx flags after new ADC data has been processed and set any TX flags when there is new processed data available for the codec ISR. It is the responsibility of the codec interrupt service routine to set the RX flag for valid data received from the RX DMA buffer, or to clear the TX flag after transmit data has been copied to the TX DMA buffer.

Figure 22. 21065L/AD1819A VSR Flag-Set-Clear Transfer Scheme



Scheme Used with the 21065L RS232 Monitor Demonstration Programs

When the SPORT1 transmit DMA empties the transmit buffer, a SPORT1 transmit interrupt occurs. The DSP routine that was executed elsewhere would set the TX Request if new data is available. If the variable TX Request > 0 , then the SPORT1 interrupt service routine loads the data from the DSP processed User TX Buffer into the TXDMA Buffer; otherwise, the TX Buffer is loaded with 0s. After the TX Buffer is loaded, the DMA is automatically re-initialized to transmit the new data in the TX Buffer. With this structure in place, the user needs to only put data in the User TX Buffer, and then set TX Request to 1, to send data to the CODEC.

The receive portion of the CODEC interface can be designed in a similar way. The DMA for SPORT1's receive register is configured to load the Rx DMA Buffer. When the RX Buffer is full, a SPORT interrupt is forced and the Rx Request variable is set if there is valid ADC data. The DSP routine is conditionally called if the RX Request bit variable is set. If the variable > 0 then the contents of the RX Buffer is written into the User RX Buffer, and the RX Request is cleared. The DMA is reinitialized to fill the RX Buffer again.

Assembly Language Rx Ready Flag Code Example For an 8 kHz ADC/DAC Sample Rate, with 48 kHz Frame Rate

The following example shows how the DSP routine and the Codec interrupt service routine would only process ADC data whenever there is new ADC data available. The Codec ISR sets the rx flags whenever new ADC data is available. The DSP routine clears the rx flags after that data has been processed. Since we are running at 8 kHz, the DSP routine would only be executed once out of every six interrupts, since:

48 kHz Audio Frame Rate / 8 kHz Sample Rate = 6 AC'97 Audio Frames Per Sample

```
.VAR          RX_left_flag;       /* DSP algorithm only processed when these bits are set */
.VAR          RX_right_flag;
--------------
/* SPORT1 Rx Interrupt Service Routine . . .  section where ADC data is received */

check_AD1819_ADC_left:
       BTST r0 by M_Left_ADC;             /* Check Master left ADC valid bit */
       IF sz JUMP check_AD1819_ADC_right; /* If valid data then save ADC sample */
          r6 = dm(rx_buf + LEFT);         /* get Master 1819 left channel input sample */
          r6 = lshift r6 by 16;           /* shift up to MSBs to preserve sign in 1.31 format */
          dm(Left_Channel_In) = r6;       /* save to data holder for processing */
          r4 = 1;
          dm(RX_left_flag) = r4;     /* if we have a new left sample, let the DSP routine know */

check_AD1819_ADC_right:
       BTST r0 by M_Right_ADC;            /* Check Master right ADC valid bit */
       IF sz jump user_applic;            /* If valid data then save ADC sample */
          r6 = dm(rx_buf + RIGHT);        /* get Master 1819 right channel input sample */
          r6 = lshift r6 by 16;           /* shift up to MSBs to preserve sign in 1.31 format */
          dm(Right_Channel_In) = r6;      /* save to data holder for processing */
          r4 = 1;
          dm(RX_right_flag) = r4;   /* if we have a new right sample, let the DSP routine know */
 --------------
/* DSP routine section where RX flag variables are cleared */
Echo:                                     /* process both channel inputs */
       /* get input samples from data holders */
       r0 = dm(Left_Channel_In);          /* left input sample */
       r1 = dm(Right_Channel_In);         /* right input sample */
       r1 = ashift r1 by -1;              /* scale signal by 1/2 for equal mix */
       r2 = ashift r2 by -1;              /* scale signal by 1/2 for equal mix */
       r1 = r2 + r1;                      /* 1/2xL(n) + 1/2 xR(n) */

       L6 = dm(delay_time);

       /* tap output of circular delay line */
       r3 = dm(i6, 0);                    /* point to d-th tap and put in data register */
                                          /* fetch address with no update */
       /* add delayed signal together with original signal */
       r1 = ashift r1 by -1;              /* scale input signal by 1/2 for equal mix */
       r3 = ashift r3 by -1;              /* scale delayed signal by 1/2 for equal mix */
       r4 = r3 + r1;                      /* 1/2xL(n) + 1/2 xR(n) */

       /* write output samples to AD1819 Master Codec channels */
       r4 = ashift r4 by 2;               /* turn up the volume a little */
       dm(Left_Channel_Out)  = r4;        /* left output sample */
       dm(Right_Channel_Out) = r4;        /* right output sample */

       /* put input sample into tap-0 of delay line,post-modify address after storage of input */
       dm(i6, -1) = r1;                   /* put value from register r1 into delay line */
                                          /* and decrement address by -1 */
       r4 = 0;
       dm(RX_left_flag) = r4;      /* clear RX_left_flag since we have processed incoming data*/
       dm(RX_right_flag) = r4;     /* clear RX_rightflag since we have processed incoming data*/
       rts;                               /* Return from Subroutine */
```

6.6 Processing 16-bit Data In 1.31 Fractional Format Or IEEE Floating Point Format

Data that is received or transmitted in the SPORT1 ISR is in a binary, 2's complement format. The DSP interprets the data in fractional format, where all #s are between -1 and 0.9999999. Initially, the serial port places the data into internal memory in data bits D0 to D15. In order to process the fractional data in 1.31 format, the processing routine first shifts the data up by 16 bits so that it is left-justified in the upper data bits D16 to D31. This is necessary to take advantage of the fixed-point multiply/accumulator's fractional 1.31 mode, as well as offer an easy reference for converting from 1.31 fractional to floating point formats. This also guarantees that any quantization errors resulting from the computations will remain well below the 16 bit result and thus below the AD1819A DAC Noise Floor. After processing the data, the DSP shifts the 1.31 result down by 16-bits so that the data is truncated to a 1.15 number. This 1.15 fractional result is then sent to the AD1819A. Below are example instructions to demonstrate shifting of data before and after the processing of data on the Master AD1819 left channel:

32-bit Fixed Point Processing

```
r1 = dm(rx_buf + 3);              /* get AD1819A left channel input sample */
r1= lshift r1 by 16;              /* shift up to MSBs to preserve sign */
dm(Left_Channel)=r1;              /* save to data holder for processing */

/*   Process data here, data is processed in 1.31 format */

r15 = dm(Left_Channel);           /* get channel 1 output result */
r15 = lshift r6 by -16;           /* put back in bits 0..15 for SPORT tx */
dm(tx_buf + 3) = r15;             /* output left result to AD1819A Slot 3 */
```

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 AD1819A 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);
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) = r8;
```

REFERENCES

The following sources contributed information to this applications note:

1. *ADSP-21065L SHARC User's Manual*, Analog Devices, Inc.,
(82-001833-01, Prentice-Hall, September 1998)

1. *ADSP-2106x SHARC User's Manual*, Analog Devices, Inc,.
Second Edition (82-000795-03, Prentice-Hall, July 1995)

2. *AD1819A Data Sheet*, Analog Devices, Inc,.
(C3261-8-3/98, 1998)

3. *Audio Codec 97 Component Specification, Revision 1.03*, Intel Corporation,
2200 Mission College Blvd, Santa Clara, CA 95052 (September 15, 1996)

4. *EE-54 - How to Use the AD1819A Variable Sample Rate Support,*
Analog Devices Inc., http://www.analog.com/techsupt/application_notes/application_notes.html#3, (5/21/98)

(We at Analog Devices express our sincere gratitude to Al Clark of Danville Signal Processing [email address: aclark@danvillesignal.com] for his contributions regarding the use of Ring Buffers to process AD1819A data at fractional sample rates, along with his helpful hints in setting up the serial port DMA parameters for RX-interrupt-based processing at variable sample rates)

7. ADSP-21065L / AD1819A DSP Driver Description

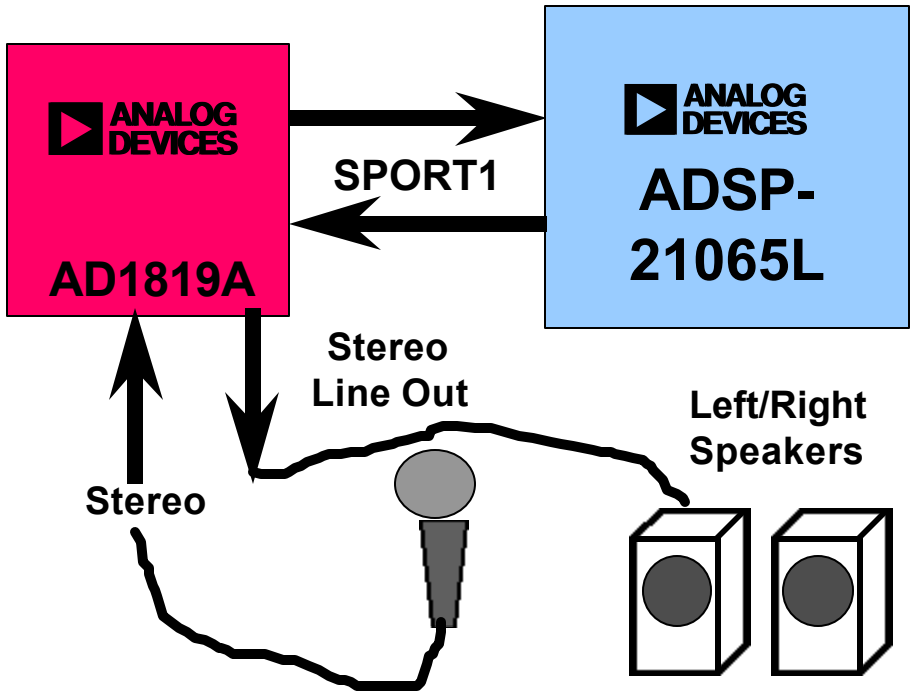


Figure 23. 21065L EZ-LAB Audio Loopback Example

The DSP source listings for AD1819A initialization and audio processing, shown in Appendix A, can be a general starting point for developing ADSP-21065L code using the AD1819A. The ADSP-21065L example program initializes the DSP serial port to communicate with the AD1819A Serial Port interface, and then perform a 'talkthru' function of audio data to and from the AD1819A. No DSP processing is performed after initialization. The only operation being performed is the fetching of data received from the AD1819 Sigma Delta ADCs and loopback the same data out to the AD1819 DACs.

The ADSP-21065L/AD1819a EZ-LAB Drivers in Appendix A are organized into the following sections:

1. 21065L EZ-LAB System Initialization Routine (for SPORT1 Tx Interrupt Audio Processing)
2. AD1819A Initialization Routine (for SPORT1 Tx Interrupt Audio Processing)
3. How To Reset The AD1819A Via DSP Control With A Flag Out Pin
4. SPORT1 Clear Routine For Use With The Target65L RS232 / VDSP Debug Monitor
5. 21065L SPORT1 RX Interrupt Service Routine For Audio Processing Using The 'ADC Valid Bits' Method For DAC Transmission
6. Installing A TX Interrupt Service Routine After Codec Register Initialization
7. Power Cycling The ADCs And DACs For Left/Right Sample Pair Synchronization With Variable Sample Rates Less Than 48 kHz
8. Variable Sample Rate Tx Interrupt Service Routine For Audio Processing - 'ADC Valid Bits' Method For DAC Transmission.
9. Variable Sample Rate Tx Interrupt Service Routine For Audio Processing - DAC Transmission Based On DAC Request Bits with ADC/DAC Ring Buffers
10. Example Ring Buffer Implementation For The 'DAC Request Bits' Method
11. Variable Sample Rate Using Both The RX And TX Interrupts For Audio Processing, 'DAC Request Bits' Version
12. Example RX ISR For Three Daisy-Chained AD1819As
13. ADSP-21065L Interrupt Vector Table
14. Visual DSP (21065L EZ-LAB) Linker Description File.

The 21065L DSP example performs the following sequence of operations to establish AD1819A communications and process audio data:

AD1819A Codec Driver Sequence Of Operations

1. **Initialize 65L DSP system stuff such as timers, flag pins, SDRAM, DAGs…**
2. **Initialize Serial Port 1 Registers**
3. **Program DMA Controller for Serial Port 1 Tx and Rx DMA chaining**
4. **Turn on Serial Port 1 and enable SPORT1 transmit interrupts**
5. **Reset the AD1819A (Optional, and required if RESET is tied to a DSP Flag Pin)**
6. **Wait for Codec to come 'on-line', and set up codec in Slot-16 Mode**
7. **Program desired AD1819A registers**
8. **Shut off SPORT1 transmit interrupts, enable SPORT1 receive interrupts (only applies if processing audio data from the SPORT Rx Interrupt)**
9. **Start processing AD1819A audio data**

7.1 List of AD1819A Reference Drivers Available From Analog Devices

Below is a listing of all the currently available 21065L EZ-LAB AD1819A drivers provided as reference from Analog Devices. These examples are fully downloadable as standalone applications via the Target65L RS232-VDSP Debugger. To obtain these reference drivers, you can submit an email to dsp.support@analog.com, or call our DSP hotline at 1-800-ANALOGD. You can also look for 65L EZ-LAB AD1819A drivers on the Analog Devices FTP server at:
ftp://ftp.analog.com/pub/dsp/audio/65L_ezlab/

Assembly Language Drivers

- Fixed 48 kHz, ADC Valid Bits Method, SPORT1 Receive Interrupt-based driver
- Variable Sample Rate, ADC Valid Bits Method, SPORT1 Receive Interrupt-based, 5-word Rx DMA buffer & 5 RX Slots Enabled, 16-word Tx DMA buffer & 16 Tx Slots Enabled
- Variable Sample Rate, DAC Request Bits Method, ADC/DAC Ring Buffers, SPORT1 Receive Interrupt-based, 5-word Rx DMA buffer & 5 RX Slots Enabled, 16-word Tx DMA buffer & 16 Tx Slots Enabled
- Variable Sample Rate, ADC Valid Bits Method, SPORT1 Transmit Interrupt-based, 7-word Tx DMA buffer & 7 TX Slots Enabled, 5-word Rx DMA buffer & 5 Rx Slots Enabled
- Variable Sample Rate, DAC Request Bits Method, ADC/DAC Ring Buffers, SPORT1 Transmit Interrupt-based, 7-word Tx DMA buffer & 7 TX Slots Enabled, 5-word Rx DMA buffer & 5 Rx Slots Enabled
- Variable Sample Rate, ADC Valid Bits Method, both SPORT1 Transmit & Receive Interrupt-based, 7-word Tx DMA buffer & 7 TX Slots Enabled, 5-word Rx DMA buffer & 5 Rx Slots Enabled
- Variable Sample Rate, DAC Request Bits Method, ADC/DAC Ring Buffers, both SPORT1 Transmit & Receive Interrupt-based, 7-word Tx DMA buffer & 7 TX Slots Enabled, 5-word Rx DMA buffer & 5 Rx Slots Enabled

C-Compiler-based Drivers

- Fixed 48 kHz, ADC Valid Bits Method, SPORT1 Receive Interrupt-based driver
- Variable Sample Rate, ADC Valid Bits Method, SPORT1 Transmit Interrupt-based, 7-word Tx DMA buffer & 7 TX Slots Enabled, 5-word Rx DMA buffer & 5 Rx Slots Enabled
- Variable Sample Rate, ADC Valid Bits Method, SPORT1 Receive Interrupt-based, 16-word Tx DMA buffer & 16 TX Slots Enabled, 5-word Rx DMA buffer & 5 Rx Slots Enabled
- Variable Sample Rate, ADC Valid Bits Method, both SPORT1 Transmit & Receive Interrupt-based, 5-word Tx DMA buffer & 5 TX Slots Enabled, 5-word Rx DMA buffer & 5 Rx Slots Enabled
- Variable Sample Rate, DAC Request Bits Method, both SPORT1 Transmit & Receive Interrupt-based, 5-word Tx DMA buffer & 5 TX Slots Enabled, 5-word Rx DMA buffer & 5 Rx Slots Enabled

APPENDIX A:
Source Code Listing for 21065L EZ-LAB Audio Driver (Visual DSP 4.x Project Files)

21065L EZ-LAB System Initialization Routine

(For SPORT1 TX ISR-based processing)

```
/***   INIT_065L_EZLAB.ASM   ************************************************
*                                                                             *
*         ADSP-21065L EZ-LAB Initialization and Main Program Shell            *
*         Developed using the ADSP-21065L EZ-LAB Evaluation Platform          *
*                                                                             *
*         This routine contains function calls and routines to initialize the *
*         state of the 21065L, program the DMA controller, initialize the AD1819a *
*         and configure the SDRAM interface.  DSP algorithm buffer initializations *
*         are also called withing this routine.                               *
*                                                                             *
*                                                 John Tomarakos              *
*                                                 ADI DSP Applications Group  *
*                                                 Revision 2.0                *
*                                                 12/17/98                    *
*                                                                             *
*******************************************************************************/

/* ADSP-21065L System Register bit definitions */
#include  "def21065l.h"
#include  "new65Ldefs.h"

.GLOBAL   _main;
.GLOBAL   Init_DSP;
.EXTERN   Init_65L_SDRAM_Controller;
.EXTERN   Blink_LEDs_Test;
.EXTERN   Program_SPORT1_Registers;
.EXTERN   Program_DMA_Controller;
.EXTERN   AD1819_Codec_Initialization;


/*-----------------------------------------------------------------------*/
.SEGMENT/dm dm_data;

.var sine4000[4000] = "sinetbl.dat";           /* optional, used for generating sine tone to AD1819A DACs */
.global audio_frame_timer;
.var audio_frame_timer = 0;                    /* 48kHz timer variable */

.endseg;

/*-----------------------------------------------------------------------*/
.segment /pm pm_code;

_main:    call Init_65L_SDRAM_Controller;      /* Initialize External Memory */
          call Program_SPORT1_Registers;       /* Initialize SPORT1 for codec communications */
          call Program_DMA_Controller;         /* Start Serial Port 1 tx and rx DMA Transfers */
          call AD1819_Codec_Initialization;    /* Initialize & program AD1819 */
          call Init_DAGs;

          IRPTL = 0x00000000;                  /* clear pending interrupts */
          bit set imask SPT1I;                 /* start audio processing, re-enable SPORT1 tx int */

          call Blink_LEDs_Test;                /* Are We Alive? */

wait_forever:
          call wait_flag1;
          bit tgl ustat1 0x3F;                 /* toggle flag 4-9 LEDs */
          dm(IOSTAT)=ustat1;
          jump wait_forever;
```

```
/*---------------------------------------------------------------------------------------- *
 *                                                                                        *
 *                                        Subroutines                                     *
 *                                                                                        *
 *---------------------------------------------------------------------------------------- */
wait_flag1:
         /* wait for flag 1 button press and release */
         if flag1_in jump wait_flag1;                  /* wait for button press */
release:
         if not flag1_in jump release;                 /* wait for button release */
         rts;

/*-----------------------------------------------------------------------------------------*/
/* Note: This routine is first called at the Reset Vector in the Interrupt Vector Table */
/*-----------------------------------------------------------------------------------------*/
Init_DSP:
         /* code to blink flag 4 */
         ustat1=0x3F;                                  /* flags 4 thru 9 are outputs for LEDs */
         dm(IOCTL)=ustat1;
         bit set ustat1 0x3F;                          /* toggle flag 4-9 LEDs */
         dm(IOSTAT)=ustat1;                            /* turn on all LEDs */
         bit clr mode2 FLG2O | FLG1O | FLG0O;          /* flag 3, 2 & 0 inputs  */

         bit set mode2 IRQ1E | IRQ2E;                  /* irqx edge sensitive */
         bit clr mode2 IRQ0E;                          /* keep irq1 to level sensitive for UART */
         IRPTL = 0x00000000;                           /* clear pending interrupts */
         bit set mode1 IRPTEN | NESTM;                 /* enable global interrupts & nesting */
         bit set imask IRQ0I | IRQ1I | IRQ2I;          /* irq1 & irq2 enabled, keep irq0 enabled for UART */
         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;

Init_DAGs:
         B1=sine4000;
         L1=4000;
         I1=sine4000;
         M1=50;

         B2=sine4000;
         L2=4000;
         I2=sine4000;
         M2=40;
         RTS;

.endseg;
```

AD1819A Initialization Routine (For SPORT Tx Interrupt Processing)

```
/***   AD1819a_initialization.ASM   ****************************************
*                                                                            *
*       AD1819A/ADSP-21065L SPORT1 Initialization Driver                     *
*       Developed using the ADSP-21065L EZ-LAB Evaluation Platform           *
*                                                                            *
*       This version sets up codec communication for Variable Sample Rate    *
*       Support.  After codec register programming, the ADCs and DACs are    *
*       powered down and back up again, so left/right valid bits and DAC     *
*       requests occur simultaneously in the same audio frame.               *
*                                                                            *
*       For efficient handling of Tag bits/ADC valid/DAC requests, the codec  *
*       ISR is processes using the SPORT1 TX (vs RX) interrupt. The SPORT1 TX *
*       interrupt is used to first program the AD1819A registers, with only  *
*       an RTI at that vector location. After codec initialization, the SPORT1 *
*       TX ISR jump label is installed, replacing an 'RTI' instruction, so that *
*       normal codec audio processing begins at that point.                  *
*                                                                            *
*                                               John Tomarakos               *
*                                               ADI DSP Applications Group   *
*                                               Revision 3.0                 *
*                                               04/29/99                     *
*                                                                            *
******************************************************************************/

/* ADSP-21060 System Register bit definitions */
#include  "def21065l.h"
#include  "new65Ldefs.h"

.EXTERN            spt1_svc;
.GLOBAL            Program_SPORT1_Registers;
.GLOBAL            Program_DMA_Controller;
.GLOBAL            AD1819_Codec_Initialization;
.GLOBAL            tx_buf;
.GLOBAL            rx_buf;
.EXTERN            Clear_All_SPT1_Regs;

/* AD1819 Codec Register Address Definitions */
#define            REGS_RESET                  0x0000
#define            MASTER_VOLUME               0x0200
#define            RESERVED_REG_1              0x0400
#define            MASTER_VOLUME_MONO          0x0600
#define            RESERVED_REG_2              0x0800
#define            PC_BEEP_Volume              0x0A00
#define            PHONE_Volume                0x0C00
#define            MIC_Volume                  0x0E00
#define            LINE_IN_Volume              0x1000
#define            CD_Volume                   0x1200
#define            VIDEO_Volume                0x1400
#define            AUX_Volume                  0x1600
#define            PCM_OUT_Volume              0x1800
#define            RECORD_SELECT               0x1A00
#define            RECORD_GAIN                 0x1C00
#define            RESERVED_REG_3              0x1E00
#define            GENERAL_PURPOSE             0x2000
#define            THREE_D_CONTROL_REG         0x2200
#define            RESERVED_REG_4              0x2400
#define            POWERDOWN_CTRL_STAT         0x2600
#define            SERIAL_CONFIGURATION        0x7400
#define            MISC_CONTROL_BITS           0x7600
#define            SAMPLE_RATE_GENERATE_0      0x7800
#define            SAMPLE_RATE_GENERATE_1      0x7A00
#define            VENDOR_ID_1                 0x7C00
#define            VENDOR_ID_2                 0x7E00

/* Mask bit selections in Serial Configuration Register for
   accessing registers on any of the 3 codecs */
#define            MASTER_Reg_Mask             0x1000
#define            SLAVE1_Reg_Mask             0x2000
#define            SLAVE2_Reg_Mask             0x4000
#define            MASTER_SLAVE1               0x3000
```

```
#define             MASTER_SLAVE2               0x5000
#define             MASTER_SLAVE1_SLAVE2        0x7000

/* Macros for setting Bits 15, 14 and 13 in Slot 0 Tag Phase */
#define             ENABLE_VFbit_SLOT1_SLOT2    0xE000
#define             ENABLE_VFbit_SLOT1 0xC000

/* AD1819 TDM Timeslot Definitions */
#define             TAG_PHASE                   0
#define             COMMAND_ADDRESS_SLOT        1
#define             COMMAND_DATA_SLOT           2
#define             STATUS_ADDRESS_SLOT         1
#define             STATUS_DATA_SLOT            2
#define             LEFT                        3
#define             RIGHT                       4

#define             AD1819_RESET_CYCLES         60
                    /* ad1819 RESETb spec = 1.0(uS) min */
                    /* 60(MIPs) = 16.67 (nS) cycle time, therefore >= 40 cycles */

#define             AD1819_WARMUP_CYCLES        60000
                    /* ad1819 warm-up = 1.0(mS) */
                    /* 60(MIPs) = 16.67 (nS) cycle time, therefore >= 40000 cycles */
/*-----------------------------------------------------------------------------------------------*/

.segment /dm    dm_codec;

.var rx_buf[5];    /* SPORT1 receive DMA buffer */

                   /* SPORT1 transmit DMA buffer */
.var tx_buf[7] =    ENABLE_VFbit_SLOT1_SLOT2,   /* set valid bits for slot 0, 1, and 2 */
                    SERIAL_CONFIGURATION,       /* serial configuration register address */
                    0xFF80,                     /* initially set to 16-bit slot mode for ADI SPORT compatibility */
                    0x0000,                     /* stuff other slots with zeros for now */
                    0x0000,
                    0x0000,
                    0x0000;
                    /* slots 5 and 6 are dummy slots, to allow enough time in the TX ISR to go     */
                    /* get rx slots 4 & 5 data in same audio frame as the ADC valid tag bits.      */
                    /* This is critical for slower sample rates, where you may not have valid data */
                    /* every rx audio frame. So you want to make sure there is valid right         */
                    /* channel data in the same rx DMA buffer fill as the detection of an ADC      */
                    /* valid right bit.  These extra slots are required ONLY for fs < 48 kHz.      */

.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* receive tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 7, 1, 0;      /* transmit tcb */

/* Codec register initializations */
/* Refer to AD1819A Data Sheet for register bit assignments */
#define   Select_LINE_INPUTS          0x0404    /* LINE IN - 0X0404, Mic In - 0x0000 */
#define   Select_MIC_INPUT            0x0000
#define   Line_Level_Volume           0x0000    /* 0 dB for line inputs */
#define   Mic_Level_Volume            0x0F0F
#define   Sample_Rate                 23456

.var Init_Codec_Registers[34] =
         MASTER_VOLUME,               0x0000,                    /* Master Volume set for no attenuation */
         MASTER_VOLUME_MONO,          0x8000,                    /* Master Mono volume is muted */
         PC_BEEP_Volume,              0x8000,                    /* PC volume is muted */
         PHONE_Volume,                0x8008,                    /* Phone Volume is muted */
         MIC_Volume,                  0x8008,                    /* MIC Input analog loopback is muted */
         LINE_IN_Volume,              0x8808,                    /* Line Input analog loopback is muted */
         CD_Volume,                   0x8808,                    /* CD Volume is muted */
         VIDEO_Volume,                0x8808,                    /* Video Volume is muted */
         AUX_Volume,                  0x8808,                    /* AUX Volume is muted */
         PCM_OUT_Volume,              0x0808,                    /* PCM out from DACs is 0db gain for both channels */
         RECORD_SELECT,               Select_LINE_INPUTS,        /* Record Select on Line Inputs for L/R channels */
         RECORD_GAIN,                 Line_Level_Volume,         /* Record Gain set for 0 dB on both L/R channels */
         GENERAL_PURPOSE,             0x0000,                    /* 0x8000, goes through 3D circuitry */
         THREE_D_CONTROL_REG,         0x0000,                    /* no phat stereo */
         MISC_CONTROL_BITS,           0x0000,                    /* use SR0 for both L & R ADCs & DACs,repeat sample */
         SAMPLE_RATE_GENERATE_0,      Sample_Rate,               /* user selectable sample rate */
         SAMPLE_RATE_GENERATE_1,      48000;                     /* Sample Rate Generator 1 not used in this example */
```

```
.var Codec_Init_Results[34] =
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0,
        0,                  0;

.endseg;

.SEGMENT /pm       pm_code;

/* ---------------------------------------------------------------------------------------- */
/* Sport1 Control Register Programming                                                      */
/* Multichannel Mode, dma w/ chain, early fs, act hi fs, fall edge, no pack, data=16/big/zero   */
/* ---------------------------------------------------------------------------------------- */

Program_SPORT1_Registers:
        /* This is required for disabling SPORT config for EZLAB RS232 debugger */
        CALL Clear_All_SPT1_Regs;           /* Clear and Reset SPORT1 and DMAs */

        /* sport1 receive control register */
        R0 = 0x0F8C40F0;                    /* 16 chans, int rfs, ext rclk, slen = 15, sden & schen enabled */
        dm(SRCTL1) = R0;                    /* sport 0 receive control register */

        /* sport1 receive frame sync divide register */
        R0 = 0x00FF0000;                    /* SCKfrq(12.288M)/RFSfrq(48.0K)-1 = 0x00FF */
        dm(RDIV1) = R0;

        /* sport1 transmit control register */
        R0 = 0x001C00F0;                    /* 1 cyc mfd, data depend, slen = 15, sden & schen enabled */
        dm(STCTL1) = R0;                    /* sport 0 transmit control register */

        /* sport1 receive and transmit multichannel word enable registers */
        R0 = 0x0000001F;                    /* enable receive channels 0-4 */
        dm(MRCS1) = R0;
        R0 = 0x0000007F;                    /* enable transmit channels 0-6 */
        dm(MTCS1) = R0;

        /* sport1 transmit and receive multichannel companding enable registers */
        R0 = 0x00000000;                    /* no companding */
        dm(MRCCS1) = R0;                    /* no companding on receive */
        dm(MTCCS1) = R0;                    /* no companding on transmit */

        RTS;

/*-------------------------------------------------------------------------------*/
/*                         DMA Controller Programming For SPORT1                 */
/*-------------------------------------------------------------------------------*/

Program_DMA_Controller:
        r1 = 0x0001FFFF;                    /* cpx register mask */

        /* sport1 dma control tx chain pointer register */
        r0 = tx_buf;
        dm(xmit_tcb + 7) = r0;              /* internal dma address used for chaining */
        r0 = 1;
        dm(xmit_tcb + 6) = r0;              /* DMA internal memory DMA modifier */
        r0 = 7;
        dm(xmit_tcb + 5) = r0;              /* DMA internal memory buffer count */
        r0 = xmit_tcb + 7;                  /* get DMA chaining internal mem pointer containing tx_buf address */
```

```
         r0 = r1 AND r0;                     /* mask the pointer */
         r0 = BSET r0 BY 17;                 /* set the pci bit */
         dm(xmit_tcb + 4) = r0;              /* write DMA transmit block chain pointer to TCB buffer */
         dm(CPT1A) = r0;                     /* transmit block chain pointer, initiate tx0 DMA transfers */

         /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
         /* - Note: Tshift0 & TX0 will be automatically loaded with the first 2 values in the   -*/
         /* - Tx buffer. The Tx buffer pointer ( II3 ) will increment by 2x the modify value    -*/
         /* - ( IM3 ).                                                                           -*/
         /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

         /* sport1 dma control rx chain pointer register */
         r0 = rx_buf;
         dm(rcv_tcb + 7) = r0;               /* internal dma address used for chaining */
         r0 = 1;
         dm(rcv_tcb + 6) = r0;               /* DMA internal memory DMA modifier */
         r0 = 5;
         dm(rcv_tcb + 5) = r0;               /* DMA internal memory buffer count */
         r0 = rcv_tcb + 7;
         r0 = r1 AND r0;                     /* mask the pointer */
         r0 = BSET r0 BY 17;                 /* set the pci bit */
         dm(rcv_tcb + 4) = r0;               /* write DMA receive block chain pointer to TCB buffer */
         dm(CPR1A) = r0;                     /* receive block chain pointer, initiate rx0 DMA transfers */

         RTS;
/* ------------------------------------------------------------------------------------ */
/*                            AD1819A Codec Initialization                              */
/* ------------------------------------------------------------------------------------ */

AD1819_Codec_Initialization:
         bit set imask SPT1I;                /* enable sport0 x-mit interrupt */

Wait_Codec_Ready:                            /* Wait for CODEC Ready State */
         R0 = DM(rx_buf + 0);                /* get bit 15 status bit from AD1819 tag phase slot 0 */
         R1 = 0x8000;                        /* mask out codec ready bit in tag phase */
         R0 = R0 AND R1;                     /* test the codec ready status flag bit */
         IF EQ JUMP Wait_Codec_Ready;        /* if flag is lo, continue to wait for a hi */

         idle;                               /* wait for a couple of TDM audio frames to pass */
         idle;

Initialize_1819_Registers:
         i4 = Init_Codec_Registers;                  /* pointer to codec initialization commands */
         r15 = ENABLE_VFbit_SLOT1_SLOT2;             /* enable v-frame bit, & slots 1&2 valid data bits */
         LCNTR = 17, DO Codec_Init UNTIL LCE;
            dm(tx_buf + TAG_PHASE) = r15             /* set valid slot bits in tag phase for slots 0,1,2 */
            r1 = dm(i4, 1);                          /* fetch next codec register address */
            dm(tx_buf + COMMAND_ADDRESS_SLOT) = r1;  /* put codec register address into tx slot 1 */
           r1 = dm(i4, 1);                           /* fetch register data contents */
           dm(tx_buf + COMMAND_DATA_SLOT) = r1;      /* put fetched codec register data into tx slot 2 */
Codec_Init: idle;                                    /* wait until TDM frame is transmitted */

/*-------------------------------------------------------------------------------------------*/
/* Verify integrity of AD1819a indexed control registers to see if communication was successful */
/*-------------------------------------------------------------------------------------------- */
/* This section of codes is for debugging/verification of AD1819 registers.  Theses          */
/* instructions initiate codec read requests of registers shown in the Init_Codec_Registers  */
/* buffer.  The results of the read requests are placed in an output buffer called           */
/* Codec_Init_Results, in which even DSP memory addresses contain the AD1819A register      */
/* address, and the DSP's odd address in the buffer contains the register data for the       */
/* AD1819A address.The AD1819A registers can then be verified with a JTAG emulator or the   */
/* 65L RS232 VDSP debug monitor program. This section of code can be removed after debug.    */
/*-------------------------------------------------------------------------------------------*/

verify_reg_writes:
         i4 = Init_Codec_Registers;
         m4 = 2;
         i5 = Codec_Init_Results;
         r15 = ENABLE_VFbit_SLOT1;                   /* enable valid frame bit, and slots 1 data bits */
         LCNTR = 17, Do ad1819_register_status UNTIL LCE;
            dm(tx_buf + TAG_PHASE) = r15;            /* set valid slot bits in tag phase for slots 0,1,2 */
            r1 = dm(i4,2);                           /* get indexed register address that is to be inspected */
            r2 = 0x8000;                             /* set bit #15 for read request in command address word */
```

```
            r1 = r1 OR r2;                              /* OR read request with the indirect register value */
            dm(tx_buf + COMMAND_ADDRESS_SLOT) = r1;     /* send out command address timeslot */
            idle;                                       /* wait 2 audio frames to go by, latency in getting data */
            idle;
            r3 = dm(rx_buf + STATUS_ADDRESS_SLOT);
            dm(i5,1) = r3;
            r3 = dm(rx_buf + STATUS_DATA_SLOT);         /* fetch requested indexed register data */
            dm(i5,1) = r3;                              /* store to results buffer */
ad1819_register_status:
            nop;

         /* For variable sample rate support, you must powerdown and powerback up the ADCs and DACs
         so that the incoming ADC data and DAC requests occur in left/right pairs */
PowerDown_DACs_ADCs:
         idle;
         r15 = ENABLE_VFbit_SLOT1_SLOT2;                /* enable valid frame bit, and slots 1&2 valid data bits */
         dm(tx_buf + TAG_PHASE) = r15;                  /* set valid slot bits in tag phase for slots 0, 1 , 2 */
         r0=POWERDOWN_CTRL_STAT;
         dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
         r0=0x0300;                                     /* power down all DACs/ADCs */
         dm(tx_buf + COMMAND_DATA_SLOT) = r0;
         idle;
         idle;

         LCNTR = AD1819_RESET_CYCLES-2, DO reset_loop UNTIL LCE;
reset_loop:   NOP;                                      /* wait for the min RESETb lo spec time */

         idle;
         r15 = ENABLE_VFbit_SLOT1_SLOT2;                /* enable valid frame bit, and slots 1&2 valid data bits */
         dm(tx_buf + TAG_PHASE) = r15;                  /* set valid slot bits in tag phase for slots 0, 1 , 2 */
         r0=POWERDOWN_CTRL_STAT;                        /* address to write to  */
         dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
         r0=0;                                          /* power up all DACs/ADCs */
         dm(tx_buf + COMMAND_DATA_SLOT) = r0;
         idle;
         idle;

         LCNTR = AD1819_WARMUP_CYCLES-2, DO warmup_loop2 UNTIL LCE;
warmup_loop2:  NOP;                                     /* wait for AD1819 warm-up */

/* ------------------------------------------------------------------------------------------ */

         bit clr imask SPT1I;                           /* disable sport1 xmit  */

Install_ISR_SPORT1_Tx_ISR:
         /* Use SPORT1 TX interrupt service function call for audio processing */
         /* install the transmit interrupt function call to replace the initial RTI instruction */
         /* "JUMP Process_AD1819_Audio_Samples" instruction into PX (0x063E 0000 xxxx). */
         /* xxxx = address of Process_AD1819_Audio_Samples */
         PX2 = 0x063e0000;                              /* Upper 32 bit Opcode for 'JUMP xxxx' instruction */
         PX1 = Process_AD1819_Audio_Samples;            /* Lower 16 bits of Opcode contain jump address */
         PM(spt1_svc) = PX;                             /* copy to 0x34 - SPORT1 interrupt vect location */

         RTS;                                           /* End of AD1819A Initialization */

/* ------------------------------------------------------------------------------------------ */
 .endseg;
```

How To Reset The AD1819A Via DSP Control With A Flag Out Pin

The following code fragment demonstrates how to reset the codec through output flag control to fulfill the minimum AD1819A reset and warm up requirements. This example was used with the AD1819 MAFE (on the SHARC 21062 EZ-LAB) to reset the AD1819. Note that this code example assumes the use of a 33 MIPs SHARC DSP. For the 21065L, the #defines should be modified to assume the use of 60 MHz.

```
#define AD1819_RESET_CYCLES  34
                             /* ad1819 RESETb spec = 1.0(uS) min */
                             /* 33.3(MIPs) = 30.0(nS) cycle time, therefore >=    33.3 cycles */

#define AD1819_WARMUP_CYCLES 33334
                             /* ad1819 warm-up = 1.0(mS) */
                             /* 33.3(MIPs) = 30.0(nS) cycle time, therefore >= 33333.3 cycles */

/* ------------------------------------------------------------ */

Reset_1819:
         /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
         /* - ad1819a mafe pin assignments:                            - */
         /* -            FLAGOUT2 = RESETb = ad1819 reset              - */
         /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

         /* CODEC Reset */
         BIT CLR ASTAT FLG2;                                             /* clear i/o flag #2 to assert RESETb pin */
         LCNTR = AD1819_RESET_CYCLES-2 , DO rsetloop UNTIL LCE ;
rsetloop: NOP;                                                           /* wait for the min RESETb */

         /* CODEC Warm-Up */
         BIT SET ASTAT FLG2;                                             /* set i/o flag #2 to deassert RESETb pin */
         LCNTR = AD1819_WARMUP_CYCLES-2 , DO warmloop UNTIL LCE;
warmloop: NOP;                                                           /* wait for warm-up */

         RTS;
```

SPORT1 Clear Routine For Use With The Target65L RS232 / VDSP Debug Monitor

```
/* ////////////////////////////////////////////////////////////////////////////////
/   ROUTINE TO CLEAR AND RESET ALL SPORT1 REGISTERS                                /
/   This routine may be required for certaing AD1819A demos when using the 21065L EZ-LAB   /
/   RS232 Debug Monitor program.  The 21065L EZ-LAB boot EPROM Monitor kernel on power-up  /
/   executes a routine that programs the SPORT1 Control and DMA registers to       /
/   communicate with the AD1819A for the example supplied EZ-LAB demo programs.    /
/                                                                                  /
/   When invoking the 65L VDSP RS232 Debugger, SPORT1 DMA is already active in     /
/   multichannel mode with DMA chaining.  If we wish to leave the SPORT TDM and DMA       /
/   channel configuration the same (i.e. 5 active channels and 5-word DMA buffers),       /
/   we are usually still able to reprogram the DMA controller to point to our own         /
/   own codec buffers with no side effects.  However, if we need to change any SPORT      /
/   control parameters such as the number of active TDM channels and DMA buffer sizes,    /
/   then the active EPROM Monitor SPORT TDM configuration on powerup of the EZ-LAB board  /
/   will affect the re-programming of the SPORT within a downloaded DSP executable.       /
/                                                                                  /
/   Since the monitor program has already activated the SPORT1 registers after a board    /
/   reset, the side effect that occurs (when re-writing over the SPORT1 control           /
/   registers) is that MCM DMA data transfers are mistakenly restarted             /
/   without the full programming of all the SPORT parameters.  Also, the TX and    /
/   RX buffers may be partially full or full, and can affect the DMA controller's         /
/   ability to correctly DMA in/out SPORT data to/from internal memory.  What results     /
/   is a failed link between the AD1819a and SPORT1 in user-modified code, because        /
/   transmitted and received data is sent or received in different timeslots and misalign  /
/   in the SPORT DMA buffers.                                                      /
/                                                                                  /
/   This routine simply clears all SPORT1 ctrl and DMA registers back to their     /
/   default states so that we can reconfigure it for our AD1819a application.      /
/                                                                                  /
/   John Tomarakos                                                                 /
/   ADI DSP Applications                                                           /
/   Rev 1.0                                                                        /
/   4/30/99                                                                        /
//////////////////////////////////////////////////////////////////////////////// */

/* ADSP-21060 System Register bit definitions */
#include  "def21065l.h"
#include  "new65Ldefs.h"
.GLOBAL   Clear_All_SPT1_Regs;

.SEGMENT /pm       pm_code;

Clear_All_SPT1_Regs:
         IRPTL = 0x00000000;                  /* clear pending interrupts */
         bit clr imask SPT1I;
         R0 = 0x00000000;
         dm(SRCTL1) = R0;                     /* sport1 receive control register */
         dm(RDIV1) = R0;                      /* sport1 receive frame sync divide register */
         dm(STCTL1) = R0;                     /* sport 0 transmit control register */
         dm(MRCS1) = R0;                      /* sport1 receive multichannel word enable register */
         dm(MTCS1) = R0;                      /* sport1 transmit multichannel word enable register */
         dm(MRCCS1) = R0;                     /* sport1 receive multichannel companding enable register */
         dm(MTCCS1) = R0;                     /* sport1 transmit multichannel companding enable register */

         /* reset SPORT1 DMA parameters back to the Reset Default State */
         R1 = 0x1FFFF;      dm(IIR1A) = R1;
         R1 = 0x0001;       dm(IMR1A) = R1;
         R1 = 0xFFFF;       dm(CR1A)  = R1;
         R1 = 0x1FFFF;      dm(CPR1A) = R1;
         R1 = 0x1FFFF;      dm(GPR1A) = R1;
         R1 = 0x1FFFF;      dm(IIT1A) = R1;
         R1 = 0x0001;       dm(IMT1A) = R1;
         R1 = 0xFFFF;       dm(CT1A)  = R1;
         R1 = 0x1FFFF;      dm(CPT1A) = R1;
         R1 = 0x1FFFF;      dm(GPT1A) = R1;
         RTS;
.ENDSEG;
```

SPORT1 Receive Interrupt Service Routine For Audio Processing Using The 'ADC Valid Bits' Method For DAC Transmission

```
/*****************************************************************************************************************************************

                                        AD1819A SPORT1 RX INTERRUPT SERVICE ROUTINE

Receives MIC1 input from the AD1819A via SPORT0 and then transmits the audio data back out to the AD1819A Stereo DACs/Line Outputs.
This routine sends DAC data based on valid ADC data in a given audio frame.

This ISR version assumes the use of the default 48kHz pro audio sample rate, in which data is valid for every audio frame, if the tx and
rx DMA buffers are 5 words with 5 channels enabled on the TDM interface.  Therefore, TAG slot info and ADC valid bit synchronization is
not as critical, since the tag bits and ADC valid bits are being set by the AD1819a and the DSP every time there is a new audio frame
(and thus a new interrupt) Therefore, the RX Interrupt can be used for audio processing after codec initialization.  This makes it
somewhat easier to initialize the codec, while saving the user the extra overhead and code space for programming the codec to use it's
variable sample rate features. If the user wishes to use the RX interrupt for variable sample rates < 48 kHz, then the tx DMA buffer
should be 16 words, the DMA tx Count register equal to 16, and channels 0-15 enabled on the tx TDM interface, so that we have plenty of
time to write to the tx TAG slot.
                                                                                                                        JT
                                                                                                                        ADI DSP Applications
                                                                                                                        Rev 3.0
                                                                                                                        12/17/98

*****************************************************************************************************************************************

   Serial Port 1 Receive Interrupt Service Routine performs arithmetic computations on SPORT1 receive
   data buffer (rx_buf) and sends results to SPORT1 transmit data buffer (tx_buf)

 rx_buf[5] - DSP SPORT receive buffer
 Slot # Description                          DSP Data Memory Address
 ------ -------------------------------------  ------------------------------------------------
 0       AD1819A Tag Phase                   DM(rx_buf + 0) = DM(rx_buf + TAG_PHASE)
 1      Status Address Port                  DM(rx_buf + 1) = DM(rx_buf + STATUS_ADDRESS_SLOT)
 2      Status Data Port                     DM(rx_buf + 2) = DM(rx_buf + STATUS_DATA_SLOT)
 3      Master PCM Capture (Record) Left Chan.  DM(rx_buf + 3) = DM(rx_buf + LEFT)
 4      Master PCM Capture Right Channel     DM(rx_buf + 4) = DM(rx_buf + RIGHT)

 tx_buf[5] - DSP SPORT transmit buffer
 Slot # Description                          DSP Data Memory Address
 ------ -------------------------------------  ------------------------------------------------
 0       ADSP-2106x Tag Phase                DM(tx_buf + 0) = DM(tx_buf + TAG_PHASE)
 1      Command Address Port                 DM(tx_buf + 1) = DM(rx_buf + COMMAND_ADDRESS_SLOT)
 2      Command Data Port                    DM(tx_buf + 2) = DM(rx_buf + COMMAND_DATA_SLOT)
 3      Master PCM Playback Left Channel     DM(tx_buf + 3) = DM(rx_buf + LEFT)
 4      Master PCM Playback Right Channel    DM(tx_buf + 4) = DM(rx_buf + RIGHT)

*****************************************************************************************/

/* ADSP-21060 System Register bit definitions */
#include  "def21065l.h"
#include  "new65Ldefs.h"

/* AD1819 TDM Timeslot Definitions */
#define            TAG_PHASE                  0
#define            COMMAND_ADDRESS_SLOT       1
#define            COMMAND_DATA_SLOT          2
#define            STATUS_ADDRESS_SLOT        1
#define            STATUS_DATA_SLOT           2
#define            LEFT                       3
#define            RIGHT                      4

/* Left and Right ADC valid Bits used for testing of valid audio data in current TDM frame */
#define            M_Left_ADC                 12
#define            M_Right_ADC                11

.GLOBAL            Process_AD1819_Audio_Samples;
.GLOBAL            Left_Channel;
.GLOBAL            Right_Channel;
.EXTERN            tx_buf;
.EXTERN            rx_buf;

.segment /dm    dm_data;
```

```
/* AD1819a stereo-channel data holders - used for DSP processing of audio data recieved from codec */
.VAR            Left_Channel;
.VAR            Right_Channel;

.endseg;

.segment /pm pm_code;

Process_AD1819_Audio_Samples:
        r0 = 0x8000;                            /* Clear all AC97 link Audio Output Frame slots */
        dm(tx_buf + TAG_PHASE) = r0;            /* and set Valid Frame bit in slot 0 tag phase */
        r0 = 0;
        dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
        dm(tx_buf + COMMAND_DATA_SLOT) = r0;
        dm(tx_buf + LEFT) = r0;
        dm(tx_buf + RIGHT) = r0;

Check_ADCs_For_Valid_Data:
        r0 = dm(rx_buf + TAG_PHASE);            /* Get ADC valid bits from tag phase slot */
        r1 = 0x1800;                            /* Mask other bits in tag */
        r2 = r0 and r1;

Set_TX_Slot_Valid_Bits:
        r1 = dm(tx_buf + TAG_PHASE);            /* frame/addr/data valid bits */
        r3 = r2 or r1;                          /* set tx valid bits based on recieve tag info */
        dm(tx_buf + TAG_PHASE) = r3;

Check_AD1819_ADC_Left:
        BTST r0 by M_Left_ADC;                  /* Check Master left ADC valid bit */
        IF sz JUMP Check_AD1819_ADC_Right;      /* If valid data then save ADC sample */
           r6 = dm(rx_buf + LEFT);              /* get Master 1819 left channel input sample */
           r6 = lshift r6 by 16;                /* shift up to MSBs to preserve sign */
           dm(Left_Channel) = r6;               /* save to data holder for processing */

Check_AD1819_ADC_Right:
        BTST r0 by M_Right_ADC;                 /* Check Master right ADC valid bit */
        IF sz RTI;                              /* If valid data then save ADC sample */
           r6 = dm(rx_buf + RIGHT);             /* get Master 1819 right channel input sample */
           r6 = lshift r6 by 16;                /* shift up to MSBs to preserve sign */
           dm(Right_Channel) = r6;              /* save to data holder for processing */

/* ---------------------------------------------------------------------------------------------------- */
/*      *** Insert DSP Algorithms Here ***                                                              */
/*                                                                                                      */
/*      Input L/R Data Streams - DM(Left_Channel) DM(Right_Channel)                                     */
/*      Output L/R Results     - DM(Left_Channel) DM(Right_Channel)                                     */
/*                                                                                                      */
/*      These left/right data holders are used to pipeline data through    multiple modules, and       */
/*      can be removed if the dsp programmer needs to save instruction cycles                          */
/*      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~        */
/*      Coding TIP:                                                                                     */
/*      The samples from the AD1819A are 16-bit and are in the lower 16    bits of the the 32-bit      */
/*      word.  They are shifted to the most significant bit positions in order to preserve the         */
/*      sign of the samples when they are converted to floating point numbers. The values are          */
/*      also scaled to the range +/-1.0 with the integer to float conversion                           */
/*      (f0 = float r0 by r1).                                                                          */
/*                                                                                                      */
/*      To convert between our assumed 1.31 fractional number and IEEE floating point math,            */
/*      here are some example assembly instructions ...                                                */
/*                                                                                                      */
/*              r1 = -31     <-- scale the sample to the range of +/-1.0                               */
/*              r0 = DM(Left_Channel);                                                                  */
/*              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) = r8;                                                                  */
/* ---------------------------------------------------------------------------------------------------- */

user_dsp_applic:

/* ---- DSP processing is finished, now playback results to AD1819 ---- */
```

```
Playback_Audio_Data:
         r15 = dm(Left_Channel);                        /* get channel 1 output result */
         r15 = lshift r15 by -16;                       /* put back in bits 0..15 for SPORT tx */
         dm(tx_buf + LEFT) = r15;                       /* output left result to Master AD1819 Slot 3 */

         r15 = dm(Right_Channel);                       /* get channel 2 output result */
         rti(db);
         r15 = lshift r15 by -16;                       /* put back in bits 0..15 for SPORT tx */
         dm(tx_buf + RIGHT) = r15;                      /* output left result to Master AD1819 Slot 3 */
/* ------------------------------------------------------------------------------------- */
.endseg;
```

Installing A TX Interrupt Routine After Codec Register Initialization

```
Install_ISR_SPORT1_Tx_ISR:
         /* Use SPORT1 TX interrupt service function call for audio processing */
         /* install the transmit interrupt function call to replace the initial RTI instruction */
         /* "JUMP Process_AD1819_Audio_Samples" instruction into PX (0x063E 0000 xxxx). */
         /* xxxx = address of Process_AD1819_Audio_Samples */
         PX2 = 0x063e0000;                              /* Upper 32 bit Opcode for 'JUMP xxxx' instruction */
         PX1 = Process_AD1819_Audio_Samples;            /* Lower 16 bits of Opcode contain jump address */
         PM(spt1_svc) = PX;                             /* copy to 0x34 - SPORT1 interrupt vect location */
```

Power Cycling The ADCs And DACs For Left/Right Sample Pair Synchronization With Variable Sample Rates < 48 KHz

```
/* For variable sample rate support, you must powerdown and powerback up the ADCs and DACs
   so that the incoming ADC data and DAC requests occur in left/right pairs */

PowerDown_DACs_ADCs:
         idle;
         r15 = ENABLE_VFbit_SLOT1_SLOT2;                        /* enable valid frame bit, and slots 1&2 valid data bits */
         dm(tx_buf + TAG_PHASE) = r15;                          /* set valid slot bits in tag phase for slots 0, 1 , 2 */
         r0=POWERDOWN_CTRL_STAT;
         dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
         r0=0x0300;                                             /* power down all DACs/ADCs */
         dm(tx_buf + COMMAND_DATA_SLOT) = r0;
         idle;
         idle;

         LCNTR = AD1819_RESET_CYCLES-2, DO reset_loop UNTIL LCE;
reset_loop:  NOP;                                               /* wait for the min RESETb lo spec time */

         idle;
         r15 = ENABLE_VFbit_SLOT1_SLOT2;                        /* enable valid frame bit, and slots 1&2 valid data bits */
         dm(tx_buf + TAG_PHASE) = r15;                          /* set valid slot bits in tag phase for slots 0, 1 , 2 */
         r0=POWERDOWN_CTRL_STAT;                                /* address to write to */
         dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
         r0=0;                                                  /* power up all DACs/ADCs */
         dm(tx_buf + COMMAND_DATA_SLOT) = r0;
         idle;
         idle;

         LCNTR = AD1819_WARMUP_CYCLES-2, DO warmup_loop2 UNTIL LCE;
warmup_loop2: NOP;                                              /* wait for AD1819 warm-up */
```

Variable Sample Rate ISR Using The TX Interrupt For Processing - 'ADC Valid Bits' Method For DAC Transmission

```
/* ********************************************************************************************
/                                                                                             /
/                           AD1819A - SPORT1 TX INTERRUPT SERVICE ROUTINE                     /
/                                                                                             /
/    Receives MIC1/Line input data from the AD1819A via SPORT1 and transmits processed audio data   /
/    back out to the AD1819A Stereo DACs/Line Outputs                                         /
/                                                                                             /
/    This SPORT1 tx ISR version uses the ADC Valid Bits to send and receive audio samples at        /
/    different rates other than the default 48 kHz.  Assuming the L/R ADCs and DACs are running     /
/    at the same sample rate, we transmit a processed sample for every newly received ADC sample.   /
/                                                                                             /
******************************************************************************************** /
                                                                                              /
  This Serial Port 1 Transmit Interrupt Service Routine performs arithmetic computations on         /
  the SPORT1 receive DMA buffer (rx_buf) and places results to SPORT1 transmit DMA buffer (tx_buf)  /
                                                                                              /
 rx_buf[5] - DSP SPORT receive buffer                                                         /
 Slot # Description                          DSP Data Memory Address                          /
 ------ ------------------------------------ ------------------------------------------------   /
 0      AD1819A Tag Phase                    DM(rx_buf + 0) = DM(rx_buf + TAG_PHASE)             /
 1      Status Address Port                  DM(rx_buf + 1) = DM(rx_buf + STATUS_ADDRESS_SLOT)   /
 2      Status Data Port                     DM(rx_buf + 2) = DM(rx_buf + STATUS_DATA_SLOT)      /
 3      Master PCM Capture (Record) Left Chan.  DM(rx_buf + 3) = DM(rx_buf + LEFT)               /
 4      Master PCM Capture Right Channel     DM(rx_buf + 4) = DM(rx_buf + RIGHT)                 /
                                                                                              /
 tx_buf[7] - DSP SPORT transmit buffer                                                        /
 Slot # Description                          DSP Data Memory Address                          /
 ------ ------------------------------------ ------------------------------------------------  /
 0      ADSP-21065L Tag Phase                DM(tx_buf + 0) = DM(tx_buf + TAG_PHASE)             /
 1      Command Address Port                 DM(tx_buf + 1) = DM(rx_buf + COMMAND_ADDRESS_SLOT)  /
 2      Command Data Port                    DM(tx_buf + 2) = DM(rx_buf + COMMAND_DATA_SLOT)     /
 3      Master PCM Playback Left Channel     DM(tx_buf + 3) = DM(rx_buf + LEFT)                  /
 4      Master PCM Playback Right Channel    DM(tx_buf + 4) = DM(rx_buf + RIGHT)                 /
 5      Dummy Slot (Not Used)                                                                /
 6      Dummy Slot (Not used)                                                                /
                                                                                              /
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

/* ADSP-21060 System Register bit definitions */
#include  "def21065l.h"
#include  "new65Ldefs.h"

/* AD1819 TDM Timeslot Definitions */
#define            TAG_PHASE                  0
#define            COMMAND_ADDRESS_SLOT       1
#define            COMMAND_DATA_SLOT          2
#define            STATUS_ADDRESS_SLOT        1
#define            STATUS_DATA_SLOT           2
#define            LEFT                       3
#define            RIGHT                      4

/* Left and Right ADC valid Bits used for testing of valid audio data in current TDM frame */
#define            M_Left_ADC                 12
#define            M_Right_ADC                11
#define            DAC_Req_Left               0x80
#define            DAC_Req_Right              0x40

.GLOBAL            Process_AD1819_Audio_Samples;
.GLOBAL            Left_Channel_In;
.GLOBAL            Right_Channel_In;
.GLOBAL            Left_Channel_Out;
.GLOBAL            Right_Channel_Out;
.GLOBAL            RX_left_flag;
.GLOBAL            RX_right_flag;
.EXTERN            tx_buf;
.EXTERN            rx_buf;
.EXTERN            fir;
```

```
.segment /dm    dm_codec;

/* AD1819a stereo-channel data holders - used for DSP processing of audio data received from codec */
.VAR      Left_Channel_In;            /* Input values from AD1819A ADCs */
.VAR      Right_Channel_In;
.VAR      Left_Channel_Out;           /* Output values for AD1819A DACs */
.VAR      Right_Channel_Out;
.VAR      Left_Channel;               /* can use for intermediate results to next filter stage */
.VAR      Right_Channel;              /* can use for intermediate results to next filter stage */
.VAR      RX_left_flag;               /* DSP algorithm only processed when these bits are set */
.VAR      RX_right_flag;
.VAR      ADC_valid_bits;

/* AC'97 audio frame/ISR counter, for debug purposes */
.VAR      audio_frame_timer = 0;

.endseg;

.segment /pm pm_code;

Process_AD1819_Audio_Samples:
          /* Build Transmit Tag Phase Slot Information */
          r0 = 0x8000;                                  /* Set Valid Frame bit 15 in slot 0 tag phase */
          dm(tx_buf + TAG_PHASE) = r0;                  /* Write tag to tx-buf ASAP before it's shifted out SPORT! */
          r0 = 0;                                       /* Clear AC97 link Audio Output Frame slots for now */
          dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
          dm(tx_buf + COMMAND_DATA_SLOT) = r0;
          dm(tx_buf + LEFT) = r0;
          dm(tx_buf + RIGHT) = r0;

check_ADCs_for_valid_data:
          r0 = dm(rx_buf + TAG_PHASE);                  /* Get ADC valid bits from tag phase slot */
          r1 = 0x1800;                                  /* Inspect for valid L/R ADC data */
          r2 = r0 and r1;                               /* Mask other bits in tag */
          dm(ADC_valid_bits) = r2;

set_tx_slot_valid_bits:
          r1 = dm(tx_buf + TAG_PHASE);                  /* set tx valid bits based on ADC valid bits info */
          r3 = r2 or r1;                                /* set left/right channel bits in tag, if required */
          dm(tx_buf + TAG_PHASE) = r3;                  /* Write tag to tx-buf ASAP before it's shifted out SPORT! */
check_AD1819_ADC_left:
          BTST r2 by M_Left_ADC;                        /* Check Master left ADC valid bit */
          IF sz JUMP check_AD1819_ADC_right;            /* If valid data then save ADC sample, otherwise continue */
             r6 = dm(rx_buf + LEFT);                    /* get Master 1819 left channel input sample */
             r6 = lshift r6 by 16;                      /* shift up to MSBs to preserve sign in 1.31 format */
             dm(Left_Channel_In) = r6;                  /* save to data holder for processing */
             r4 = 1;
             dm(RX_left_flag) = r4;                     /* if we have a new left sample, let the DSP routine know */

check_AD1819_ADC_right:
          BTST r2 by M_Right_ADC;                       /* Check Master right ADC valid bit */
          If sz jump user_applic;                       /* If valid data then save ADC sample, otherwise continue */
             r6 = dm(rx_buf + RIGHT);                   /* get Master 1819 right channel input sample */
             r6 = lshift r6 by 16;                      /* shift up to MSBs to preserve sign in 1.31 format */
             dm(Right_Channel_In) = r6;                 /* save to data holder for processing */
             r4 = 1;
             dm(RX_right_flag) = r4;                    /* if we have a new right sample, let the DSP routine know */


/* ------------------------------------------------------------------------------------------------------- */
/*        user_applic( ) - User Applications Routines                                                      */
/*        *** Insert DSP Algorithms Here ***                                                               */
/*                                                                                                         */
/*        Input L/R Data Streams - DM(Left_Channel_In)  DM(Right_Channel_In)                               */
/*        Output L/R Results     - DM(Left_Channel_Out) DM(Right_Channel_Out)                              */
/*                                                                                                         */
/*        These left/right data holders are used to pipeline data through    multiple modules, and        */
/*        can be removed if the dsp programmer needs to save instruction cycles                           */
/*        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  */
/*        Coding TIP:                                                                                      */
/*        The samples from the AD1819A are 16-bit and are in the lower 16 bits of the the 32-bit           */
/*        word.  They are shifted to the most significant bit positions in order to preserve the           */
/*        sign of the samples when they are converted to floating point numbers. The values are            */
```

```
/*        also scaled to the range +/-1.0 with the integer to float conversion                          */
/*        (f0 = float r0 by r1).                                                                       */
/*                                                                                                     */
/*        To convert between our assumed 1.31 fractional number and IEEE floating point math,          */
/*        here are some example assembly instructions ...                                              */
/*                                                                                                     */
/*                  r1 = -31     <-- scale the sample to the range of +/-1.0                           */
/*                  r0 = DM(Left_Channel);                                                             */
/*                  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) = r8;                                                             */
/* ------------------------------------------------------------------------------------------------- */

user_applic:
          /* do audio processing on input samples */
          r4 = 0x0;                              /* since we are not using the right flag, always clear */
          dm(RX_right_flag) = r4;                /* since left & right come in pairs at same fs rate, we
                                                    only need one flag (because we powered down ADCs for L/R sync).
                                                    Thus, the user can optionally remove the right flag set/clear
                                                    instructions to save cycles */
          r4 = dm(RX_left_flag);
          r4 = pass r4;
          if eq jump playback_audio_data;        /* if RX_left_flag = 1, then do audio processing */
                                                 /* filter routine will clear RX_left_flag */
do_audio_processing:
          call (pc, fir);

          /* ---- DSP processing is finished, now playback results to AD1819 ---- */

playback_audio_data:
          /* Transmit Left and Right Valid Data every time there the ADCs have valid data */
          r2 = dm(ADC_valid_bits);

tx_AD1819_DAC_left:
          BTST r2 by M_Left_ADC;                 /* Check to see if we need to send DAC right sample */
          IF sz JUMP tx_AD1819_DAC_right;        /* If valid data then transmit DAC sample */
          r15 = dm(Left_Channel_Out);            /* get channel 1 output result */
          r15 = lshift r15 by -16;               /* put back in bits 0..15 for SPORT tx */
          dm(tx_buf + LEFT) = r15;               /* output right result to AD1819a Slot 3 */

tx_AD1819_DAC_right:
          BTST r2 by M_Right_ADC;                /* Check to see if we need to send DAC right sample */
          If sz jump tx_done;                    /* If valid data then transmit DAC sample */
          r15 = dm(Right_Channel_Out);           /* get channel 2 output result */
          r15 = lshift r15 by -16;               /* put back in bits 0..15 for SPORT tx */
          dm(tx_buf + RIGHT) = r15;              /* output right result to AD1819a Slot 4 */

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 */

/* ------------------------------------------------------------------------------------------- */

.endseg;
```

Variable Sample Rate ISR Using The TX Interrupt For Processing - DAC Transmission Based On DAC Request Bits With Ring Buffers

```
/* ********************************************************************************************
/                                                                                             /
/                           AD1819A - SPORT1 TX INTERRUPT SERVICE ROUTINE                     /
/                                                                                             /
/    Receives MIC1/Line input data from the AD1819A via SPORT1 and transmits processed audio data   /
/    back out to the AD1819A Stereo DACs/Line Outputs                                         /
/                                                                                             /
/    This SPORT1 tx ISR version uses the DAC Request Bits to send and receive audio samples at      /
/    different rates other than the default 48 kHz.  Assuming the L/R ADCs and DACs are running     /
/    at the same sample rate, we transmit a processed sample based on the DAC requests from the     /
/    AD1819A while performing the processing based on the ADC valid bits.                     /
/                                                                                             /
********************************************************************************************
                                                                                              /
  This Serial Port 1 Transmit Interrupt Service Routine performs arithmetic computations on         /
  the SPORT1 receive DMA buffer (rx_buf) and places results to SPORT1 transmit DMA buffer (tx_buf)  /
                                                                                              /
 rx_buf[5] - DSP SPORT receive buffer                                                         /
 Slot # Description                          DSP Data Memory Address                          /
 ------ ------------------------------------  ------------------------------------------------   /
 0      AD1819A Tag Phase                    DM(rx_buf + 0) = DM(rx_buf + TAG_PHASE)             /
 1      Status Address Port                  DM(rx_buf + 1) = DM(rx_buf + STATUS_ADDRESS_SLOT)   /
 2      Status Data Port                     DM(rx_buf + 2) = DM(rx_buf + STATUS_DATA_SLOT)      /
 3      Master PCM Capture (Record) Left Chan.  DM(rx_buf + 3) = DM(rx_buf + LEFT)               /
 4      Master PCM Capture Right Channel     DM(rx_buf + 4) = DM(rx_buf + RIGHT)                 /
                                                                                              /
 tx_buf[7] - DSP SPORT transmit buffer                                                        /
 Slot # Description                          DSP Data Memory Address                          /
 ------ ------------------------------------  ------------------------------------------------  /
 0      ADSP-21065L Tag Phase                DM(tx_buf + 0) = DM(tx_buf + TAG_PHASE)             /
 1      Command Address Port                 DM(tx_buf + 1) = DM(rx_buf + COMMAND_ADDRESS_SLOT)  /
 2      Command Data Port                    DM(tx_buf + 2) = DM(rx_buf + COMMAND_DATA_SLOT)     /
 3      Master PCM Playback Left Channel     DM(tx_buf + 3) = DM(rx_buf + LEFT)                  /
 4      Master PCM Playback Right Channel    DM(tx_buf + 4) = DM(rx_buf + RIGHT)                 /
 5      Dummy Slot (Not Used)                                                                /
 6      Dummy Slot (Not used)                                                                /
                                                                                              /
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* ADSP-21060 System Register bit definitions */
#include  "def21065l.h"
#include  "new65Ldefs.h"

/* AD1819 TDM Timeslot Definitions */
#define            TAG_PHASE                  0
#define            COMMAND_ADDRESS_SLOT       1
#define            COMMAND_DATA_SLOT          2
#define            STATUS_ADDRESS_SLOT        1
#define            STATUS_DATA_SLOT           2
#define            LEFT                       3
#define            RIGHT                      4

/* Left and Right ADC valid Bits used for testing of valid audio data in current TDM frame */
#define            M_Left_ADC                 12
#define            M_Right_ADC                11
#define            DAC_Req_Left               0x80
#define            DAC_Req_Right              0x40

.GLOBAL            Process_AD1819_Audio_Samples;
.GLOBAL            Left_Channel_In;
.GLOBAL            Right_Channel_In;
.GLOBAL            Left_Channel_Out;
.GLOBAL            Right_Channel_Out;
.EXTERN            tx_buf;
.EXTERN            rx_buf;

.segment /dm    dm_codec;

/* AD1819a stereo-channel data holders - used for DSP processing of audio data received from codec */
```

```
.VAR      Left_Channel_In;
.VAR      Right_Channel_In;
.VAR      Left_Channel_Out;
.VAR      Right_Channel_Out;

/* AD1819 ADC/DAC ring buffer variables, may be required for Fractional Sample Rate Rations of 48 kHz*/
.VAR      Lchan_ring_buff[6] = 0, 0, 0, 0, 0, 0;
.VAR      Rchan_ring_buff[6] = 0, 0, 0, 0, 0, 0;
.VAR      L_input_ptr;  /* temporary storage of Index register, this saves us from using 4 DAG pointers */
.VAR      L_DAC_output_ptr;
.VAR      R_input_ptr;
.VAR      R_DAC_output_ptr;
.VAR      ADC_sample_test = 0x00000000;
.GLOBAL   Lchan_ring_buff, Rchan_ring_buff, L_input_ptr, L_DAC_output_ptr, R_input_ptr, R_DAC_output_ptr;

/* AC'97 audio frame/ISR counter, for debug purposes */
.VAR      audio_frame_timer = 0;

.endseg;

.segment /pm pm_code;

Process_AD1819_Audio_Samples:
          /* Build Transmit Tag Phase Slot Information */
          r0 = 0x00c0;                                  /* slots3 and slots 4 DAC REQ bit mask */

check_DAC_request_bits:
          /* do not overwrite contents of r1, since it is required at the end of this interrupt! */
          r1 = dm(rx_buf + STATUS_ADDRESS_SLOT);        /* Get ADC request bits from address slot */
          r2 = r1 and r0;                               /* Mask out the AD1819 Master DRRQ0 and DLRQ0 bits */
          r2 = r2 xor r0;                               /* Set active low DAC request bits to active hi */
          r2 = lshift r2 by 5;                          /* shift up so output tag info is bits 12 and 11 */

set_TX_slot_valid_bits:
          r0 = 0x8000;                                  /* Write tag to tx-buf ASAP before it's shifted out! */
          r2 = r2 or r0;                                /* set tx valid bits based on received DAC request info */
          dm(tx_buf + TAG_PHASE) = r2;                  /* Set Valid Frame & Valid Slot bits in slot 0 tag phase */
          r0 = 0;                                       /* Clear all AC97 link Audio Output Frame slots */
          dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
          dm(tx_buf + COMMAND_DATA_SLOT) = r0;
          dm(tx_buf + LEFT) = r0;
          dm(tx_buf + RIGHT) = r0;

          r0 = dm(rx_buf + TAG_PHASE);                  /* get tag information to inspect for valid L/R ADC data */
          DM(ADC_sample_test) = r0;                     /* save for conditional ring buffer input storage */

check_AD1819_ADC_left:
          BTST r0 by M_Left_ADC;                        /* Check Master left ADC valid bit */
          IF sz JUMP check_AD1819_ADC_right;            /* If valid data then save ADC sample */
             r6 = dm(rx_buf + LEFT);                    /* get Master 1819 left channel input sample */
             r6 = lshift r6 by 16;                      /* shift up to MSBs to preserve sign in 1.31 format */
             dm(Left_Channel_In) = r6;                  /* save to data holder for processing */

check_AD1819_ADC_right:
          BTST r0 by M_Right_ADC;                       /* Check Master right ADC valid bit */
          IF sz jump user_applic;                       /* If valid data then save ADC sample */
             r6 = dm(rx_buf + RIGHT);                   /* get Master 1819 right channel input sample */
             r6 = lshift r6 by 16;                      /* shift up to MSBs to preserve sign in 1.31 format */
             dm(Right_Channel_In) = r6;                 /* save to data holder for processing */

/* ------------------------------------------------------------------------------------------------- */
/*  user_applic( ) - User Applications Routines                                                      */
/*       *** Insert DSP Algorithms Here ***                                                          */
/*                                                                                                   */
/*       Input L/R Data Streams - DM(Left_Channel_In) DM(Right_Channel_In)                           */
/*       Output L/R Results     - DM(Left_Channel_Out) DM(Right_Channel_Out)                         */
/*                                                                                                   */
/*       These left/right data holders are used to pipeline data through    multiple modules, and    */
/*       can be removed if the dsp programmer needs to save instruction cycles                       */
/*       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~   */
/*       Coding TIP:                                                                                 */
/*       The samples from the AD1819A are 16-bit and are in the lower 16    bits of the the 32-bit    */
/*       word.  They are shifted to the most significant bit positions in order to preserve the      */
```

```
/*        sign of the samples when they are converted to floating point numbers. The values are          */
/*        also scaled to the range +/-1.0 with the integer to float conversion                           */
/*        (f0 = float r0 by r1).                                                                        */
/*                                                                                                      */
/*        To convert between our assumed 1.31 fractional number and IEEE floating point math,           */
/*        here are some example assembly instructions ...                                               */
/*                                                                                                      */
/*                  r1 = -31        <-- scale the sample to the range of +/-1.0                         */
/*                  r0 = DM(Left_Channel);                                                              */
/*                  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) = r8;                                                              */
/* ---------------------------------------------------------------------------------------------------- */
user_applic:
         {call (pc, Audio_Algorithm);}

         /* ---- DSP processing is finished, now playback results to AD1819 ---- */

         call Sample_Jitter_Attenuator;                 /* call this routine if running at fractional sample rates
                                                           ratios of 48 kHz, such as 44100, 8766, 23456, etc..
                                                           otherwise, comment out or remove routine entirely */

playback_audio_data:
         /* Transmit Left and Right Valid Data if Requested */
         r2=DAC_Req_Left;                               /* Check to see if Left DAC REQ? */
         r3=r1 and r2;                                  /* DAC request is active low */
         if ne jump bypass_left;                        /* if it is 1, it means we have no request, so move on */
            call (pc, left_ring_buff_out);              /* if DAC req set, then get processed o/p from ring buffer */
            r15 = dm(Left_Channel_Out);                 /* get channel 1 output result */
            r15 = lshift r15 by -16;                    /* put back in bits 0..15 for SPORT tx */
            dm(tx_buf + LEFT) = r15;                    /* output right result to AD1819a Slot 3 */

bypass_left:
         r2=DAC_Req_Right;                              /* Check to see if Right DAC REQ? */
         r3=r1 and r2;                                  /* DAC request is active low */
         if ne jump bypass_right;                       /* if it is 1, it means we have no request, so move on */
            call (pc, right_ring_buff_out);             /* if DAC req set, then get processed o/p from ring buffer */
            r15 = dm(Right_Channel_Out);                /* get channel 2 output result */
            r15 = lshift r15 by -16;                    /* put back in bits 0..15 for SPORT tx */
            dm(tx_buf + RIGHT) = r15;                   /* output right result to AD1819a Slot 4 */

bypass_right:
         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 */
```

Example Ring Buffer Implementation For The 'DAC Request Bits' Method

```
/* ------------------------------------------------------------------------ */
/* Left/Right Channel Ring Buffers Routines, for sample rate jitter attenuation */
/* ------------------------------------------------------------------------   */
/* These ring buffers may be required when using the tx interrupt for audio     */
/* processing and running at fractional sample rate rations of 48 kHz using     */
/* the 'DAC Request Bits Method' to eliminate risk or repeating or dropping     */
/* processed samples, resulting in a loss in signal quality.                    */
/*                                                                              */
/* This method is not the most efficient, but only requires 1 index register    */
/* for implementation instead of four.  The user could also use any four        */
/* available primary or secondary index registers to implement the ring         */
/* buffers, and then remove the memory pointer save/restore instructions.       */
/* ------------------------------------------------------------------------ */

.EXTERN            Lchan_ring_buff;
.EXTERN            Rchan_ring_buff;
.EXTERN            L_input_ptr;
.EXTERN            L_DAC_output_ptr;
.EXTERN            R_input_ptr;
.EXTERN            R_DAC_output_ptr;
.EXTERN            ADC_sample_test;

.segment /pm pm_code;

Init_Ring_Buffers:
         /* initialize the ring buffer input and output pointers */
         B0 = Lchan_ring_buff;
         DM(L_input_ptr) = I0;

         I0 = Lchan_ring_buff + 3;                    /* start output ptr in middle of the buffer */
         DM(L_DAC_output_ptr) = I0;

         B0 = Rchan_ring_buff;
         DM(R_input_ptr) = I0;

         I0 = Rchan_ring_buff + 3;                    /* start output ptr in middle of the buffer */
         DM(R_DAC_output_ptr) = I0;

         L0 = 6;                                      /* both input/output L/R ring buffers are 6 words deep */
         rts;

/* ////////////////////////////////////////////////////////////////////////////////////////////// */

Sample_Jitter_Attenuator:
         L0 = 6;                                      /* input and output ring buffers are 6 words deep */
         M0 = 1;
         r9 = DM(ADC_sample_test);
         BTST r9 by M_Left_ADC;                       /* Did we process a valid left sample in this ISR? */
         IF sz JUMP right_ring_buff_in;               /* If we did, store result to left ring buffer input */

left_ring_buff_in:
         R0 = DM(Left_Channel_In);
         B0 = Lchan_ring_buff;
         I0 = DM(L_input_ptr);

            DM(I0,M0) = R0;

         DM(L_input_ptr) = I0;

right_ring_buff_in:
         BTST r9 by M_Right_ADC;                      /* Did we process a valid right sample in this ISR? */
         IF sz JUMP ring_done;                        /* If we did, store result to right ring buffer input */
         R0 = DM(Right_Channel_In);
         B0 = Rchan_ring_buff;
         I0 = DM(R_input_ptr);

            DM(I0,M0) = R0;

         DM(R_input_ptr) = I0;
```

```
ring_done:
          rts;

/* ------------------------------------------------------------------------------------------- */

left_ring_buff_out:
          L0 = 6;
          M0 = 1;
          B0 = Lchan_ring_buff;
          I0 = DM(L_DAC_output_ptr);

             R0 = DM(I0,M0);

          DM(L_DAC_output_ptr) = I0;
          DM(Left_Channel_Out) = R0;

          rts;

/* ------------------------------------------------------------------------------------------- */
right_ring_buff_out:
          B0 = Rchan_ring_buff;
          I0 = DM(R_DAC_output_ptr);

             R0 = DM(I0,M0);

          DM(R_DAC_output_ptr) = I0;
          DM(Right_Channel_Out) = R0;

          rts;

/* ------------------------------------------------------------------------------------------- */

.endseg;
```

Variable Sample Rate ISR Using Both The RX and TX Interrupt For Processing - TX DAC Transmission Based On DAC Request Bits

```
/* ********************************************************************************************

                        AD1819A - SPORT1 RX & TX INTERRUPT SERVICE ROUTINES

Receives MIC1 input from the AD1819A via SPORT0 and then transmits the audio data back out to the
AD1819A Stereo DACs/Line Outputs.  This version supports the variable sample rate features of the
AD1819A, testing both the ADC valid bits and DAC request bits for transmitting and receiving data
at sample rates (selectable in 1 Hz increments) other than 48 kHz.

********************************************************************************************

   Serial Port 1 Transmit Interrupt Service Routine performs arithmetic computations on SPORT1 receive
   data buffer (rx_buf) and sends results to SPORT1 transmit data buffer (tx_buf)

 rx_buf[5] - DSP SPORT recieve buffer
 Slot # Description                          DSP Data Memory Address
 ------  ------------------------------------ ------------------------------------------------
 0       AD1819 Tag Phase                     DM(rx_buf + 0) = DM(rx_buf + TAG_PHASE)
 1       Status Address Port                  DM(rx_buf + 1) = DM(rx_buf + STATUS_ADDRESS_SLOT)
 2       Status Data Port                     DM(rx_buf + 2) = DM(rx_buf + STATUS_DATA_SLOT)
 3       Master PCM Capture (Record) Left Chan. DM(rx_buf + 3) = DM(rx_buf + LEFT)
 4       Master PCM Capture Right Channel     DM(rx_buf + 4) = DM(rx_buf + RIGHT)

 tx_buf[5] - DSP SPORT transmit buffer
 Slot #  Description                          DSP Data Memory Address
 ------  ------------------------------------ ------------------------------------------------
 0       ADSP-2106x Tag Phase                 DM(tx_buf + 0) = DM(tx_buf + TAG_PHASE)
 1       Command Address Port                 DM(tx_buf + 1) = DM(rx_buf + COMMAND_ADDRESS_SLOT)
 2       Command Data Port                    DM(tx_buf + 2) = DM(rx_buf + COMMAND_DATA_SLOT)
 3       Master PCM Playback Left Channel     DM(tx_buf + 3) = DM(rx_buf + LEFT)
 4       Master PCM Playback Right Channel    DM(tx_buf + 4) = DM(rx_buf + RIGHT)

********************************************************************************************/

/* ADSP-21060 System Register bit definitions */
#include  "def21065l.h"
#include  "new65Ldefs.h"

/* AD1819 TDM Timeslot Definitions */
#define            TAG_PHASE                  0
#define            COMMAND_ADDRESS_SLOT       1
#define            COMMAND_DATA_SLOT          2
#define            STATUS_ADDRESS_SLOT        1
#define            STATUS_DATA_SLOT           2
#define            LEFT                       3
#define            RIGHT                      4

/* Left and Right ADC valid Bits used for testing of valid audio data in current TDM frame */
#define            M_Left_ADC                 12
#define            M_Right_ADC                11
#define            DAC_Req_Left               0x80
#define            DAC_Req_Right              0x40

.GLOBAL            Process_AD1819_Audio_Input;
.GLOBAL            Playback_Audio_Data;
.GLOBAL            Left_Channel;
.GLOBAL            Right_Channel;
.GLOBAL            Left_Channel_In;
.GLOBAL            Right_Channel_In;
.GLOBAL            Left_Channel_Out;
.GLOBAL            Right_Channel_Out;
.GLOBAL            RX_left_flag;
.GLOBAL            RX_right_flag;
.EXTERN            tx_buf;
.EXTERN            rx_buf;
.EXTERN            Slapback_Echo;
.EXTERN            Stereo_Double_Tracking;
.EXTERN            effects_counter;
```

```
.segment /dm    dm_codec;

/* AD1819A stereo-channel data holders and flags-used for DSP processing of received codec audio data*/
.VAR      Left_Channel_In;                    /* Input values from AD1819A ADCs */
.VAR      Right_Channel_In;
.VAR      Left_Channel_Out;                   /* Output values for AD1819A DACs */
.VAR      Right_Channel_Out;
.VAR      Left_Channel;                       /* can use for intermediate results to next filter stage */
.VAR      Right_Channel;                      /* can use for intermediate results to next filter stage */
.VAR      RX_left_flag;                       /* DSP algorithm only processed when these bits are set */
.VAR      RX_right_flag;
.VAR      DAC_RQ;                             /* used to pass DAC request bits info from rx ISR to tx ISR */

/* define AC-97 audio frame counters for debug purposes */
.var      rx_audio_frame_timer = 0;           /* 48kHz audio frame timer variable */
.var      tx_audio_frame_timer = 0;           /* 48kHz audio frame timer variable */

.endseg;

.segment /pm pm_code;

/* ***************************************************************************************** */
/*                                                                                           */
/*                        SPORT1 RX INTERRUPT SERVICE ROUTINE                                */
/*                                                                                           */
/* ***************************************************************************************** */

Process_AD1819_Audio_Input:
         bit set mode1 SRRFL;                          /* enable background register file */
         NOP;                                          /* 1 CYCLE LATENCY FOR WRITING TO MODE1 REGISER!! */

get_DAC_request_bits:
         r1 = dm(rx_buf + STATUS_ADDRESS_SLOT);        /* Get ADC request bits from address slot */
         dm(DAC_RQ) = r1;                              /* save for SPORT1 TX ISR */

         r0 = dm(rx_buf + TAG_PHASE);                  /* get tag information to inspect for valid L/R ADC data */

check_AD1819_ADC_left:
         BTST r0 by M_Left_ADC;                        /* Check Master left ADC valid bit */
         IF sz JUMP check_AD1819_ADC_right;            /* If valid data then save ADC sample */
            r6 = dm(rx_buf + LEFT);                    /* get Master 1819 left channel input sample */
            r6 = lshift r6 by 16;                      /* shift up to MSBs to preserve sign in 1.31 format */
            dm(Left_Channel_In) = r6;                  /* save to data holder for processing */
            r4 = 1;
            dm(RX_left_flag) = r4;                     /* if we have a new left sample, let the DSP filter routine know */

check_AD1819_ADC_right:
         BTST r0 by M_Right_ADC;                       /* Check Master right ADC valid bit */
         IF sz jump user_applic;                       /* If valid data then save ADC sample */
            r6 = dm(rx_buf + RIGHT);                   /* get Master 1819 right channel input sample */
            r6 = lshift r6 by 16;                      /* shift up to MSBs to preserve sign in 1.31 format */
            dm(Right_Channel_In) = r6;                 /* save to data holder for processing */
            r4 = 1;
            dm(RX_right_flag) = r4;                    /* if we have a new right sample, let the DSP filter routine know */

/* ---------------------------------------------------------------------------------------------- */
/*  user_applic( ) - User Applications Routines                                                   */
/*       *** Insert DSP Algorithms Here ***                                                       */
/*                                                                                                */
/*       Input L/R Data Streams - DM(Left_Channel_In) DM(Right_Channel_In)                        */
/*       Output L/R Results     - DM(Left_Channel_Out) DM(Right_Channel_Out)                      */
/*                                                                                                */
/*       These left/right data holders are used to pipeline data through    multiple modules, and */
/*       can be removed if the dsp programmer needs to save instruction cycles                    */
/*       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~        */
/*       Coding TIP:                                                                              */
/*       The samples from the AD1819A are 16-bit and are in the lower 16    bits of the the 32-bit */
/*       word.  They are shifted to the most significant bit positions in order to preserve the   */
/*       sign of the samples when they are converted to floating point numbers. The values are    */
/*       also scaled to the range +/-1.0 with the integer to float conversion                     */
/*       (f0 = float r0 by r1).                                                                   */
/*                                                                                                */
```

```
/*        To convert between our assumed 1.31 fractional number and IEEE floating point math,            */
/*        here are some example assembly instructions ...                                                */
/*                                                                                                       */
/*                r1 = -31        <-- scale the sample to the range of +/-1.0                            */
/*                r0 = DM(Left_Channel);                                                                 */
/*                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) = r8;                                                                 */
/* ------------------------------------------------------------------------------------------------------ */

user_applic:
         /* since we powered down the ADCs after codec initialization, the samples should occur in
            left right pairs.  Therefore, since both the left and right channels are running at the
            same fs, when we detect a left sample, we will call our DSP routine because we will have
            both a new left sample and a new right sample */
         r4 = dm(RX_left_flag);
         r4 = pass r4;
         if eq jump rx_end;                          /* if RX_left_flag = 1, then do audio processing */
                                                     /* delay routine will clear RX_left_flag */
do_audio_processing:
         r0 = DM(effects_counter);
         r0 = pass r0;                               /* get preset mode */
         if eq call (pc, Slapback_Echo);             /* check for count == 0 */
         r0 = DM(effects_counter);                   /* check again */
         r0 = pass r0;                               /* still the same ? */
         if eq jump rx_end;                          /* bypass if not stereo */
         r0 = r0 - 1;                                /* decrement, must be stereo effect */
         if eq call (pc, Stereo_Double_Tracking);    /* check for count == 1 */

         /* ---- DSP processing is finished, now playback results to AD1819 via TX ISR ---- */

rx_end:
         r4 = 0x0;                                   /* since we are not using the right flag, always clear */
         dm(RX_right_flag) = r4;                     /* since left & right come in pairs at same fs rate, we
                                                        only need one flag */
         r0=dm(rx_audio_frame_timer);                /* get last count */
         r0=r0+1;                                    /* increment count */
         dm(rx_audio_frame_timer)=r0;                /* save updated count */

         rti(db);                                    /* return from interrupt, delayed branch */
         bit clr mode1 SRRFL;                        /* switch back to primary register set */
         NOP;                                        /* 1 CYCLE LATENCY FOR WRITING TO MODE1 REGISER!! */


/* ************************************************************************************ */
/*                                                                                      */
/*                      SPORT1 TX INTERRUPT SERVICE ROUTINE                             */
/*                                                                                      */
/* ************************************************************************************ */


Playback_Audio_Data:
         /* Build Transmit Tag Phase Slot Information */
         r0 = 0x00c0;                                /* slots3 and slots 4 DAC REQ bit mask */

check_DAC_request_bits:
         /* DAC request bits were fetched from spt1 rx ISR! */
         r1 = dm(DAC_RQ);                            /* Get ADC request bits from address slot */
         r2 = r1 and r0;                             /* Mask out the AD1819 Master DRRQ0 and DLRQ0 bits */
         r2 = r2 xor r0;                             /* Set active low DAC request bits to active hi */
         r2 = lshift r2 by 5;                        /* shift up so output tag info is bits 12 and 11 */

set_TX_slot_valid_bits:
         r0 = 0x8000;                                /* Write tag to tx-buf ASAP before it's shifted out! */
         r2 = r2 or r0;                              /* set tx valid bits based on received DAC request info */
         dm(tx_buf + TAG_PHASE) = r2;                /* Set Valid Frame & Valid Slot bits in slot0 tag phase */
         r0 = 0;                                     /* Clear all AC97 link Audio Output Frame slots */
         dm(tx_buf + COMMAND_ADDRESS_SLOT) = r0;
         dm(tx_buf + COMMAND_DATA_SLOT) = r0;
         dm(tx_buf + LEFT) = r0;
         dm(tx_buf + RIGHT) = r0;
```

```
        /* Transmit Left and Right Valid Data if Requested */
        r2=DAC_Req_Left;                            /* Check to see if Left DAC REQ? */
        r3=r1 and r2;                               /* DAC request is active low */
        if ne jump bypass_left;                     /* if it is 1, it means we have no request, so move on */

        r15 = dm(Left_Channel_Out);                 /* get channel 1 output result */
        r15 = lshift r15 by -16;                    /* put back in bits 0..15 for SPORT tx */
        dm(tx_buf + LEFT) = r15;                    /* output right result to AD1819a Slot 3 */

bypass_left:
        r2=DAC_Req_Right;                           /* Check to see if Right DAC REQ? */
        r3=r1 and r2;                               /* DAC request is active low */
        if ne jump bypass_right;                    /* if it is 1, it means we have no request, so move on */

        r15 = dm(Right_Channel_Out);                /* get channel 2 output result */
        r15 = lshift r15 by -16;                    /* put back in bits 0..15 for SPORT tx */
        dm(tx_buf + RIGHT) = r15;                   /* output right result to AD1819a Slot 4 */

bypass_right:
        r0=dm(tx_audio_frame_timer);                /* get last count */
        rti(db);                                    /* return from interrupt, delayed branch */
        r0=r0+1;                                    /* increment count */
        dm(tx_audio_frame_timer)=r0;                /* save updated count */

/* ------------------------------------------------------------------------------------- */

.endseg;
```

Example RX ISR For Three Daisy-Chained AD1819As

```
/* ------------------------------------------------------------------------------------    */
/*    Serial Port 1 Receive Interrupt Service Routine                                      */
/*    performs arithmetic computations on SPORT1 receive data buffer (rx_buf) and          */
/*    sends results to SPORT1 transmit data buffer (tx_buf)                                */
/*
         rx_buf[9] - DSP SPORT recieve buffer
         Slot #    Description                                DSP Data Memory Address
         0         AD1819A Tag Phase                          DM(rx_buf + 0)
         1         Status Address Port                        DM(rx_buf + 1)
         2         Status Data Port                           DM(rx_buf + 2)
         3         Master PCM Capture (Record) Left Channel   DM(rx_buf + 3)
         4         Master PCM Capture Right Channel           DM(rx_buf + 4)
         5         Slave 1 PCM Capture Left Channel           DM(rx_buf + 5)
         6         Slave 1 PCM Capture Right Channel          DM(rx_buf + 6)
         7         Slave 2 PCM Capture Left Channel           DM(rx_buf + 7)
         8         Slave 2 PCM Capture Right Channel          DM(rx_buf + 8)

         tx_buf[9] - DSP SPORT transmit buffer
         Slot #  Description                                  DSP Data Memory Address
         0         ADSP-2106x Tag Phase                       DM(tx_buf + 0)
         1         Command Address Port                       DM(tx_buf + 1)
         2         Command Data Port                          DM(tx_buf + 2)
         3         Master PCM Playback Left Channel           DM(tx_buf + 3)
         4         Master PCM Playback Right Channel          DM(tx_buf + 4)
         5         Slave 1 PCM Playback Left Channel          DM(tx_buf + 5)
         6         Slave 1 PCM Playback Right Channel         DM(tx_buf + 6)
         7         Slave 2 PCM Playback Left Channel          DM(tx_buf + 7)
         8         Slave 2 PCM Playback Right Channel         DM(tx_buf + 8)          */
/*                                                                                       */
/* ------------------------------------------------------------------------------------    */

Process_AD1819_Audio_Samples:
        r0 = 0x8000;                                /* Clear all AC97 link Audio Output Frame slots */
        dm(tx_buf + 0) = r0;                        /* and set Valid Frame bit in slot 0 tag phase */
        r0 = 0;
        dm(tx_buf + 1) = r0;
        dm(tx_buf + 2) = r0;
        dm(tx_buf + 3) = r0;
        dm(tx_buf + 4) = r0;
        dm(tx_buf + 5) = r0;
        dm(tx_buf + 6) = r0;
        dm(tx_buf + 7) = r0;
        dm(tx_buf + 8) = r0;

Check_ADCs_For_Valid_Data:
        r0 = dm(rx_buf);                            /* Get ADC valid bits from tag phase slot */
        r1 = 0x1f80;                                /* Mask other bits in tag */
        r2 = r0 and r1;

Set_TX_Slot_Valid_Bits:
        r1 = dm(tx_buf + 0);                        /* frame/addr/data valid bits */
        r3 = r2 or r1;                              /* set tx valid bits based on recieve tag info */
        dm(tx_buf +0) = r3;

Check_Master_ADC_Left:
         BTST r0 by M_Left_ADC;                     /* Check Master left ADC valid bit */
         IF sz JUMP Check_Master_ADC_Right;         /* If valid data then save ADC sample */
            r6 = dm(rx_buf + 3);                    /* get Master 1819 left channel input sample */
            r6= lshift r6 by 16;                    /* shift up to MSBs to preserve sign */
            dm(Master_Left_Channel)=r6;             /* save to data holder for processing */

Check_Master_ADC_Right:
         BTST r0 by M_Right_ADC;                    /* Check Master right ADC valid bit */
         IF sz JUMP Check_Slave1_ADC_Left;          /* If valid data then save ADC sample */
            r6 = dm(rx_buf + 4);                    /* get Master 1819 right channel input sample */
            r6 = lshift r6 by 16;                   /* shift up to MSBs to preserve sign */
            dm(Master_Right_Channel) = r6;          /* save to data holder for processing */

Check_Slave1_ADC_Left:
         BTST r0 by S1_Left_ADC;                    /* Check Slave 1 left ADC valid bit */
         if sz jump Check_Slave1_ADC_Right;         /* If valid data then save ADC sample */
```

```
            r6 = dm(rx_buf + 5);                        /* get Slave 1 1819 left channel input sample */
            r6 = lshift r6 by 16;                       /* shift up to MSBs to preserve sign */
            dm(Slave1_Left_Channel) = r6;               /* save to data holder for processing */

Check_Slave1_ADC_Right:
         BTST r0 by S1_Right_ADC;                       /* Check Slave 1 right ADC valid bit */
         if sz jump Check_Slave2_ADC_Left;              /* If valid data then save ADC sample */
            r6 = dm(rx_buf + 6);                        /* get Slave 1 1819 right channel input sample */
            r6 = lshift r6 by 16;                       /* shift up to MSBs to preserve sign */
            dm(Slave1_Right_Channel) = r6;              /* save to data holder for processing */

Check_Slave2_ADC_Left:
         BTST r0 by S2_Left_ADC;                        /* Check Slave 2 left ADC valid bit */
         if sz jump Check_Slave2_ADC_Right;             /* If request is made save ADC sample */
            r6 = dm(rx_buf + 7);                        /* get Slave 2 1819 left channel input sample */
            r6 = lshift r6 by 16;                       /* shift up to MSBs to preserve sign */
            dm(Slave2_Left_Channel) = r6;               /* save to data holder for processing */

Check_Slave2_ADC_Right:
         BTST r0 by S2_Right_ADC;                       /* Check Slave 2 right ADC valid bit */
         if sz rti;                                     /* If valid data then save ADC sample */
            r6 = dm(rx_buf + 8);                        /* get Slave 2 1819 right channel input sample */
            r6 = lshift r6 by 16;                       /* shift up to MSBs to preserve sign */
            dm(Slave2_Right_Channel) = r6;              /* save to data holder for processing */

         /* ------------------------------------------- */
         /* Insert Sample/Block Processing Algorithm Here */
         /* ------------------------------------------- */

Loopback_Audio_Data:
         r15 = dm(Master_Left_Channel);                 /* get channel 1 output result */
         r15 = lshift r15 by -16;                       /* put back in bits 0..15 for SPORT tx */
         dm(tx_buf + 3) = r15;                          /* output left result to Master AD1819 Slot 3 */

         r15 = dm(Master_Right_Channel);                /* get channel 2 output result */
         r15 = lshift r15 by -16;                       /* put back in bits 0..15 for SPORT tx */
         dm(tx_buf + 4) = r15;                          /* output left result to Master AD1819 Slot 3 */

         r15 = dm(Slave1_Left_Channel);                 /* get channel 3 output result */
         r15 = lshift r15 by -16;                       /* put back in bits 0..15 for SPORT tx */
         dm(tx_buf + 5) = r15;                          /* output left result to Slave1 AD1819 Slot 3 */

         r15 = dm(Slave1_Right_Channel);                /* get channel 4 output result */
         r15 = lshift r15 by -16;                       /* put back in bits 0..15 for SPORT tx */
         dm(tx_buf + 6) = r15;                          /* output left result to Slave1 AD1819 Slot 3 */

         r15 = dm(Slave2_Left_Channel);                 /* get channel 5 output result */
         r15 = lshift r15 by -16;                       /* put back in bits 0..15 for SPORT tx */
         dm(tx_buf + 7) = r15;                          /* output left result to Slave2 AD1819 Slot 3 */

         r15 = dm(Slave2_Right_Channel);                /* get channel 6 output result */
         rti(db);
         r15 = lshift r15 by -16;                       /* put back in bits 0..15 for SPORT tx */
         dm(tx_buf + 8) = r15;                          /* output left result to Slave2 AD1819 Slot 3 */
/* ---------------------------------------------------------------------------------------------*/
.endseg;
```

ADSP-21065L Interrupt Vector Table

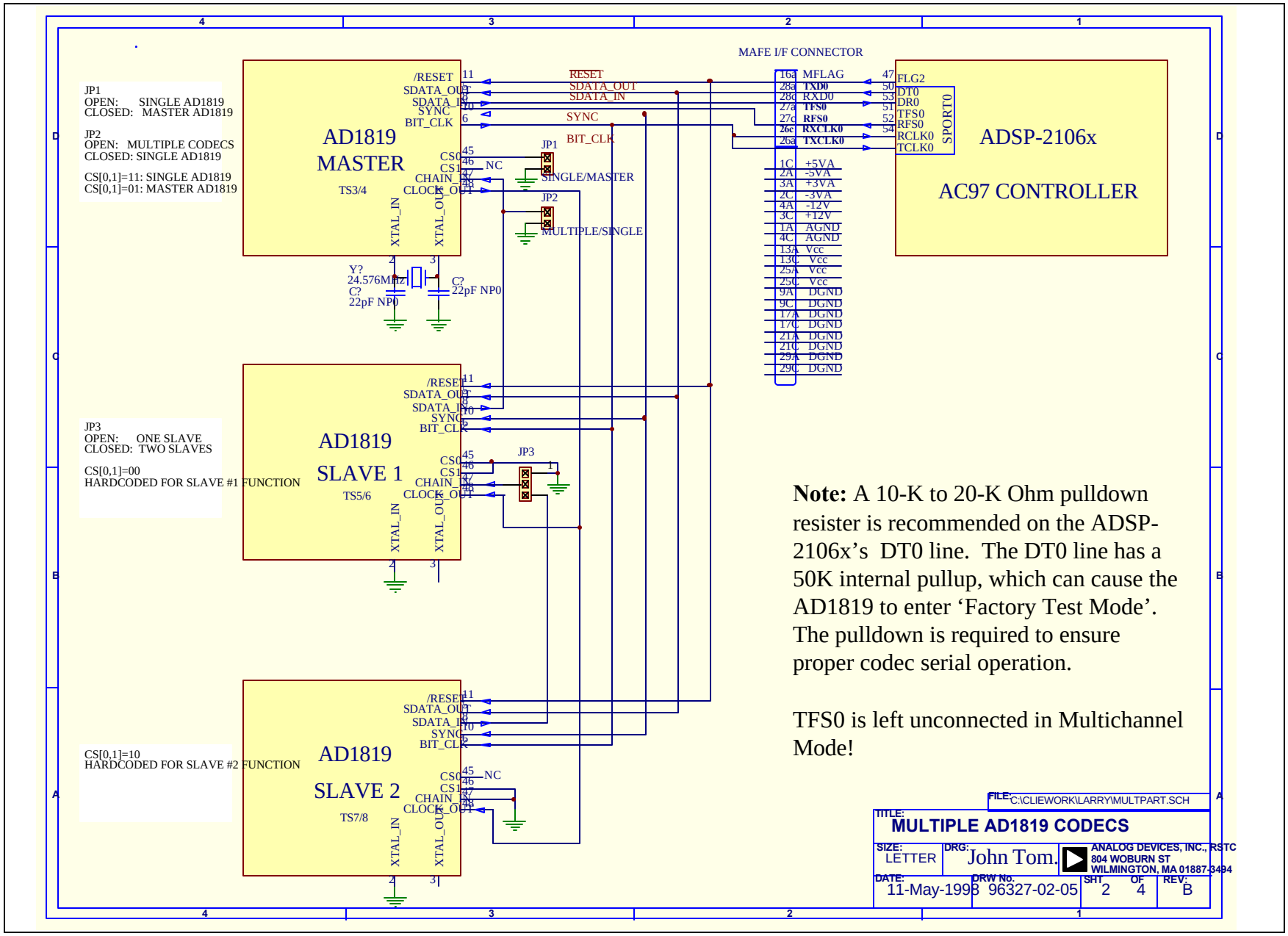
```
/* ********************************************************************************** */
/*                                                                                    */
/*                          ADSP-21065L INTERRUPT VECTOR TABLE                        */
/*                                                                                    */
/*                  For use with the 21065L EZ-LAB Evaluation Platform.               */
/*                                                                                    */
/*                                                                  (JT - 10/23/98)   */
/* ********************************************************************************** */


.EXTERN   _main;
.EXTERN   Init_DSP;
.EXTERN   Process_AD1819_Audio_Samples;

.SEGMENT/PM   isr_tbl;

/* 0x00 Reserved Interrupt */
/*                 0x00      0x01      0x02      0x03      0x04 */
/* reserved_0:     NOP;      NOP;      NOP;      NOP;      NOP; */

/* *** Reset vector *** */
/* 0x05 - reset vector starts at location 0x8005 */
rst_svc:           call Init_DSP;
                   NOP;
                   jump _main;

/* 0x08 - Reserved interrupt */
reserved_0x8:      NOP;      NOP;      NOP;      NOP;

/* 0x0C - Vector for status stack/loop stack overflow or PC stack full: */
sovfi_svc:         RTI;      RTI;      RTI;      RTI;

/* 0x10 - Vector for high priority timer interrupt: */
tmzhi_svc:         RTI;      RTI;      RTI;      RTI;

/* 0x14 - Vectors for external interrupts: */
vrpti_svc:         RTI;      RTI;      RTI;      RTI;

/* 0x18 - IRQ2 Interrupt Service Routine (ISR) */
irq2_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x1C - IRQ1 Interrupt Service Routine (ISR) */
irq1_svc:          RTI;      RTI;      RTI;      RTI;

/* *** 0x20 - IRQ0 Interrupt Service Routine (ISR) , 4 locations max *** */
 irq0_svc:         RTI;      RTI;      RTI;      RTI;

/* 0x24  - Reserved interrupt */
reserved_0x24:     NOP;      NOP;      NOP;      NOP;

/* 0x28 - Vectors for Serial Port 0 Receive A & B DMA channels 0/1 */
spr0_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x2C - Vectors for Serial Port 1 Receive A & B DMA channels 2/3 */
spr1_svc:          JUMP Process_AD1819_Audio_Samples;    RTI;      RTI;      RTI;

/* 0x30 - Vectors for Serial Port 0 Transmit A & B DMA channels 4/5 */
spt0_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x34 - Vectors for Serial Port 1 Transmit A & B DMA channels 6/7 */
spt1_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x38 - Reserved Interrupt */
reserved_0x38:     RTI;      RTI;      RTI;      RTI;

/* 0x3C - Reserved Interrupt */
reserved_0x3c:     RTI;      RTI;      RTI;      RTI;

/* 0x40 - Vector for External Port DMA channel 8 */
ep0_svc:           RTI;      RTI;      RTI;      RTI;
```

```
/* 0x44 - Vector for External Port DMA channel 9 */
ep1_svc:           RTI;      RTI;      RTI;      RTI;

/* 0x48 - Reserved Interrupt */
reserved_0x48:     RTI;      RTI;      RTI;      RTI;

/* 0x4C - Reserved Interrupt */
reserved_0x4c:     RTI;      RTI;      RTI;      RTI;

/* 0x50 - Reserved Interrupt */
reserved_0x50:     RTI;    RTI;       RTI;      RTI;

/* 0x54 - Vector for DAG1 buffer 7 circular buffer overflow */
cb7_svc:           RTI;      RTI;      RTI;      RTI;

/* 0x58 - Vector for DAG2 buffer 15 circular buffer overflow */
cb15_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x5C - Vector for lower priority timer interrupt */
tmzl_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x60 - Vector for fixed-point overflow */
fix_svc:           RTI;      RTI;      RTI;      RTI;

/* 0x64 - Floating-point overflow exception */
flt0_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x68 - Floating-point underflow exception */
fltu_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x6C - Floating-point invalid exception */
flti_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x70 - User software interrupt 0 */
sft0_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x74 - User software interrupt 1 */
sft1_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x78 - User software interrupt 2 */
sft2_svc:          RTI;      RTI;      RTI;      RTI;

/* 0x7C - User software interrupt 3 */

sft3_svc:          RTI;      RTI;      RTI;      RTI;

.ENDSEG;
```

Visual DSP Tools (21065L EZ-LAB) Linker Description File

```
// ****************************************************************************/
// *                                                                          */
// *                        21065L EZ-LAB LINKER DESCRIPTION FILE             */
// *                                                                          */
// *     For use with the 21065L EZ-LAB Evaluation Platform.  The Interrupt Table is  */
// *     split into 2 sections- low and high.  IRQ0 is removed, so that the UART     */
// *     remains functional and is not overwritten after downloading of user code    */
// *                                                                          */
// *                                                          (JT - 10/23/98)    */
// ****************************************************************************/

ARCHITECTURE(ADSP-21065L)

SEARCH_DIR( $ADI_DSP\21k\lib )

// The lib060.dlb must come before libc.dlb because libc.dlb has some 21020
// specific code and data
$LIBRARIES = lib060.dlb;

// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
$OBJECTS = $COMMAND_LINE_OBJECTS;

MAP (loopback.map)

//
// ADSP-21065L Memory Map:
//   ------------------------------------------------
//   Internal memory  0x0000 0000 to 0x0007 ffff
//   ------------------------------------------------
//                0x0000 0000 to 0x0000 00ff  IOP Regs
//                0x0000 0100 to 0x0000 01ff  IOP Regs of processor ID 001
//                0x0000 0200 to 0x0000 02ff  IOP Regs of processor ID 002
//                0x0000 0300 to 0x0000 7fff  Reserved (unusable)
//
//   Block 0       0x0000 8000 to 0x0000 9fff  Normal Word (32/48) Addresses
//                 0x0000 A000 to 0x0000 Bfff  Reserved
//   Block 1       0x0000 C000 to 0x0000 Dfff  Normal Word (32/48) Addresses
//                 0x0000 E000 to 0x0000 ffff  (Reserved)
//   Block 0       0x0001 0000 to 0x0001 3fff  Short Word address space (16-bit)
//                 0x0001 4000 to 0x0001 7fff  Reserved
//   Block 1       0x0001 8000 to 0x0001 bfff  Short Word (16) Addresses
//
//                 0x0001 C000 to 0x0001 ffff  Reserved
//
//   ------------------------------------------------
//   Multiproc memory 0x0000 0100 to 0x0000 02ff
//   ------------------------------------------------
//
//   ------------------------------------------------
//   External memory  0x0002 0000 to 0x03ff ffff
//   ------------------------------------------------
//

MEMORY
{
//       IRQ0 Interrupt 0x20 - 0x23 reserved by EZ-LAB UART Monitor Program */
         isr_tabl           { TYPE(PM RAM) START(0x00008005) END(0x0000807f) WIDTH(48) }
         pm_code            { TYPE(PM RAM) START(0x00008100) END(0x00008bff) WIDTH(48) }
         pm_data            { TYPE(PM RAM) START(0x00009400) END(0x000097ff) WIDTH(32) }
         krnl_code          { TYPE(PM RAM) START(0x00009000) END(0x000097ff) WIDTH(48) }
         dm_data            { TYPE(DM RAM) START(0x0000c000) END(0x0000dfff) WIDTH(32) }
         EMAFE_addr         { TYPE(DM RAM) START(0x01000000) END(0x01000000) WIDTH(32) }
         EMAFE_data         { TYPE(DM RAM) START(0x01000001) END(0x01000001) WIDTH(32) }
         UART_regs          { TYPE(DM RAM) START(0x01000008) END(0x0100000F) WIDTH(32) }
         codec_reset        { TYPE(DM RAM) START(0x01000010) END(0x01000010) WIDTH(32) }
         seg_dm_sdram       { TYPE(DM RAM) START(0x03000000) END(0x030ffeff) WIDTH(32) }
         krnl_ext_res       { TYPE(DM RAM) START(0x030fff00) END(0x030fffff) WIDTH(32) }
}

PROCESSOR p0
```

```
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS
    {
                    // .text output section
                    isr_tabl
                    {
                             INPUT_SECTIONS( $OBJECTS(isr_tbl) $LIBRARIES(isr_tbl))
                    } >isr_tabl

                    pm_code
                    {
                             INPUT_SECTIONS( $OBJECTS(pm_code) $LIBRARIES(pm_code))
                    } >pm_code

                    pm_data
                    {
                             INPUT_SECTIONS( $OBJECTS(pm_data) $LIBRARIES(pm_data))
                    } >pm_data

                    dm_data
                    {
                             INPUT_SECTIONS( $OBJECTS(dm_data dm_codec) $LIBRARIES(dm_data))
                    } > dm_data

                    //---------------------------------
                    //  EXTERNAL MEMORY SEGMENTS
                    // if you do not want to initialize SRAM area in executable, use SHT_NOBITS
                    // example         sdram SHT_NOBITS
                    //                 {
                    //                          INPUT_SECTIONS( $OBJECTS(segsdram))
                    //                 } > seg_dm_sdram
                    //---------------------------------

                    dm_sdram // SHT_NOBITS
                    {
                             INPUT_SECTIONS( $OBJECTS(testtone dm_delay segsdram))
                    } > seg_dm_sdram
    }
}
```



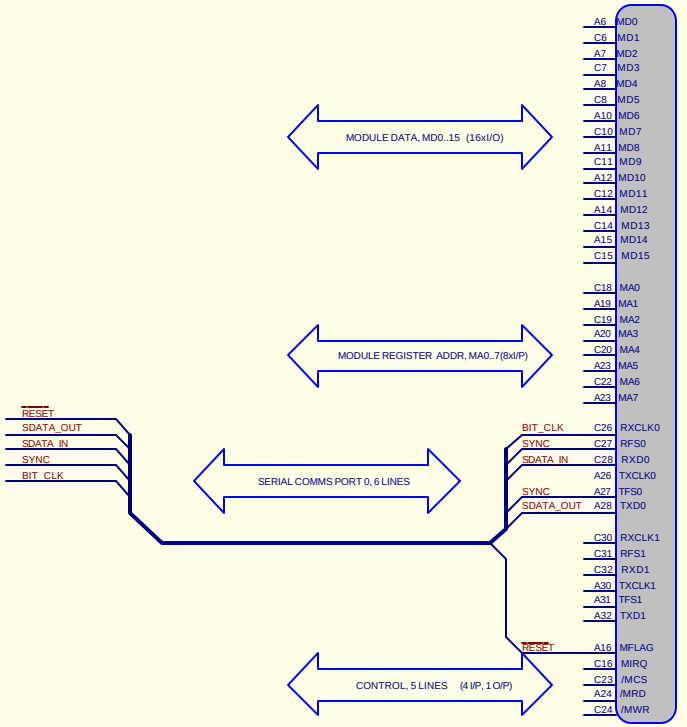
Note: A 10-K to 20-K Ohm pulldown resister is recommended on the ADSP-2106x's DT0 line. The DT0 line has a 50K internal pullup, which can cause the AD1819 to enter 'Factory Test Mode'. The pulldown is required to ensure proper codec serial operation.

TFS0 is left unconnected in Multichannel Mode!

FILE: C:\CLIEWORK\LARRY\MULTPART.SCH

TITLE: **MULTIPLE AD1819 CODECS**

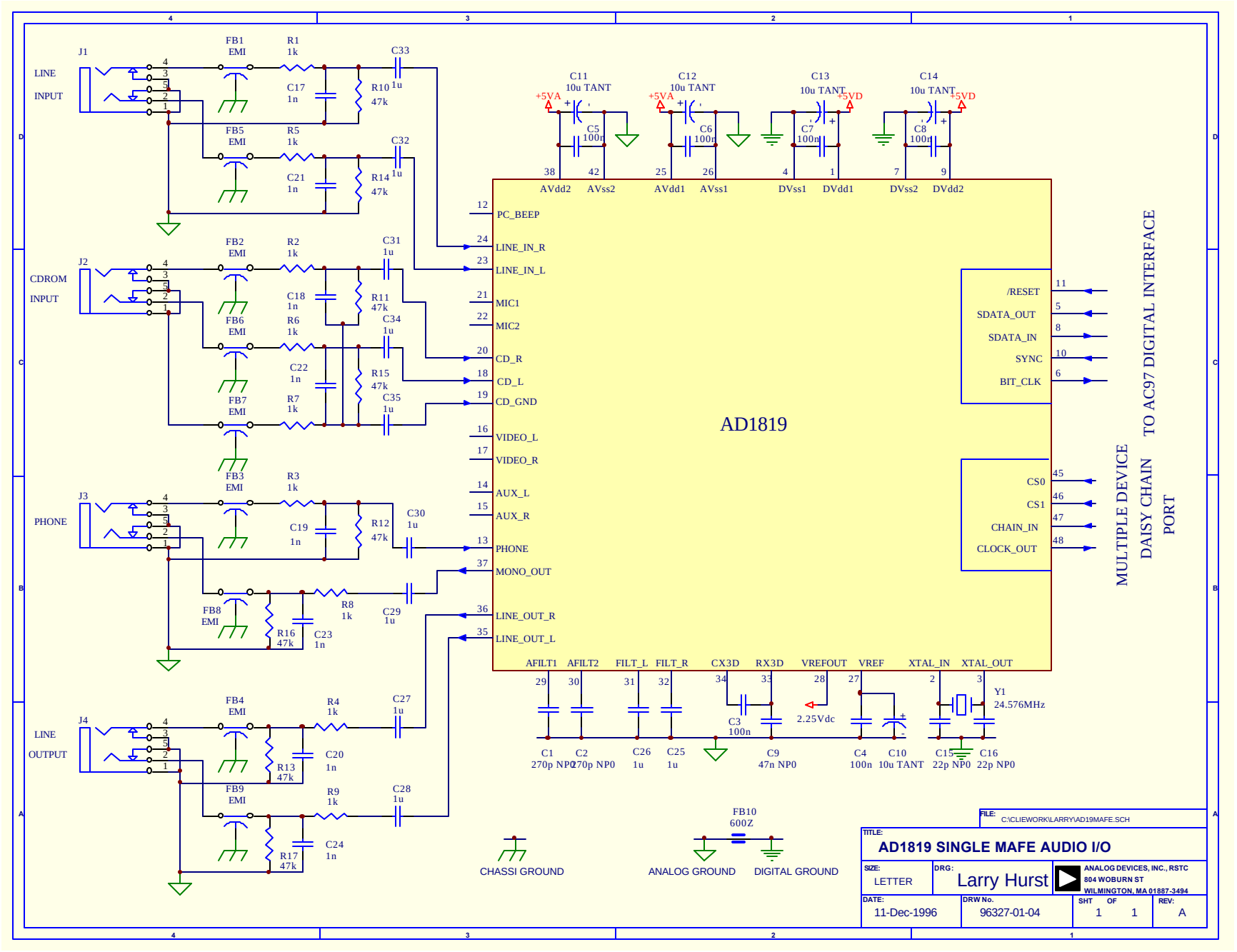
SIZE:	DRG:		
LETTER	John Tom.	ANALOG DEVICES, INC., RSTC 804 WOBURN ST WILMINGTON, MA 01887-3494	
DATE:	DRW No.	SHT OF	REV:
11-May-1998	96327-02-05	2 4	B



FILE: C:\CLIEWORK\LARRY\MAFEINTF.SCH

TITLE: **MAFE INTERFACE CONNECTIONS**

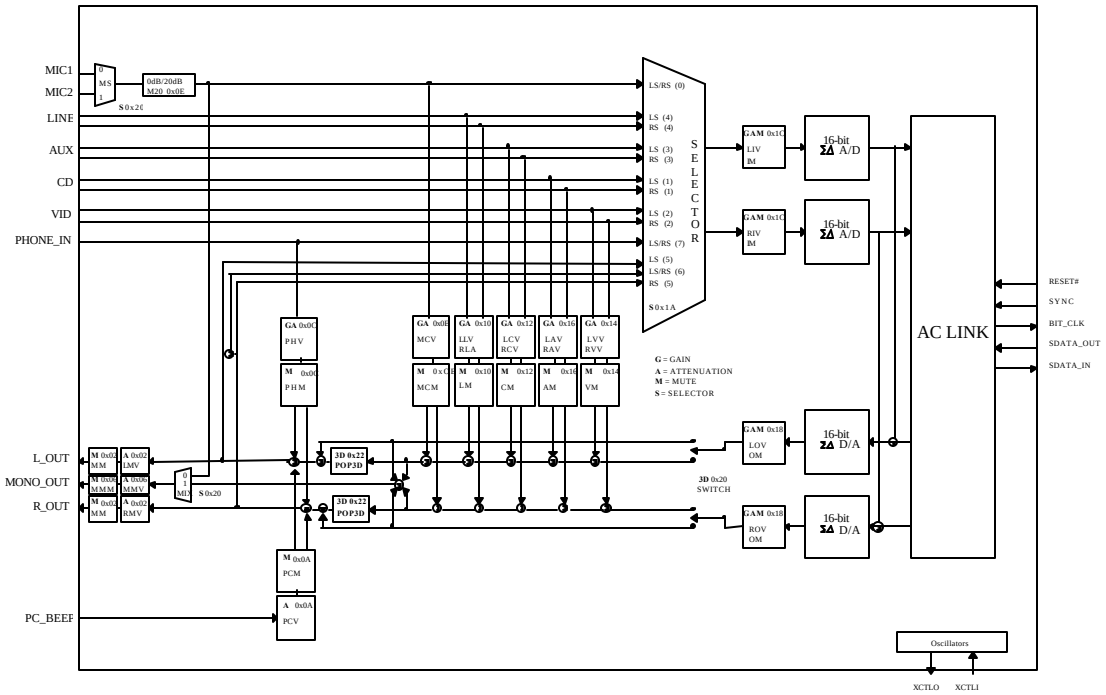
SIZE:	DRG:		
LETTER	Larry Hurst	ANALOG DEVICES, INC., RSTC 804 WOBURN ST WILMINGTON, MA 01887-3494	
DATE:	DRW No.	SHT OF	REV:
11-Dec-1996	96327-03-04	3 4	A

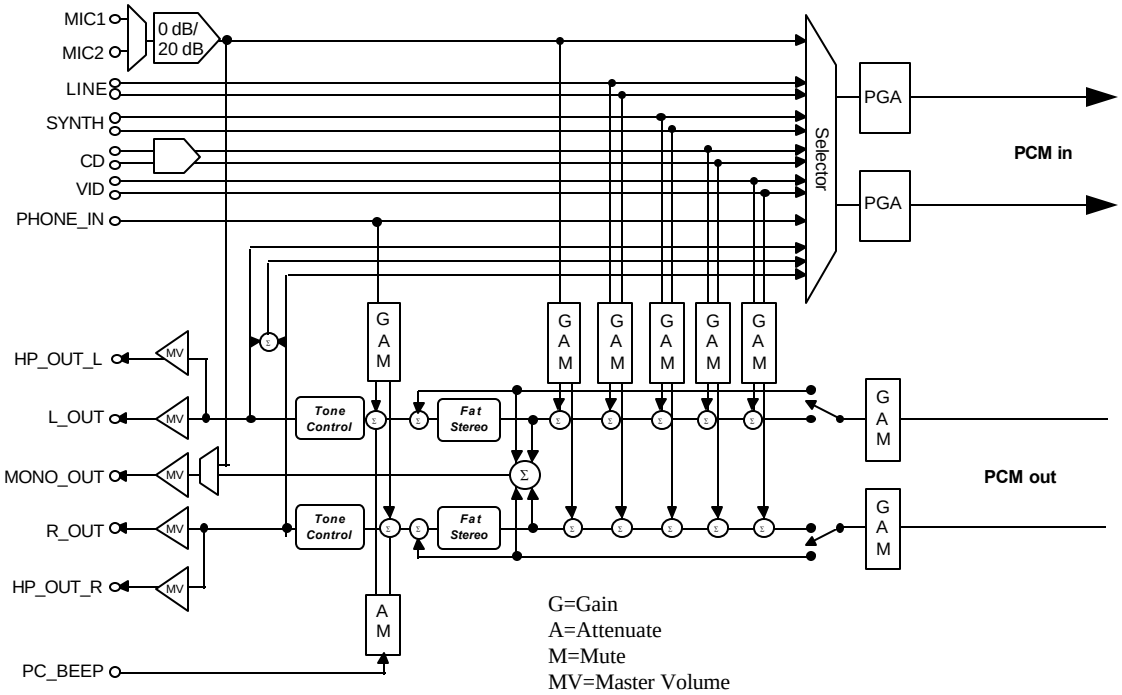

AD1819
LINE INPUT
CDROM INPUT
PHONE
LINE OUTPUT
PC_BEEP
LINE_IN_R
LINE_IN_L
MIC1
MIC2
CD_R
CD_L
CD_GND
VIDEO_L
VIDEO_R
AUX_L
AUX_R
PHONE
MONO_OUT
LINE_OUT_R
LINE_OUT_L
AVdd2
AVss2
AVdd1
AVss1
DVss1
DVdd1
DVss2
DVdd2
/RESET
SDATA_OUT
SDATA_IN
SYNC
BIT_CLK
CS0
CS1
CHAIN_IN
CLOCK_OUT
AFILT1
AFILT2
FILT_L
FILT_R
CX3D
RX3D
VREFOUT
VREF
XTAL_IN
XTAL_OUT
2.25Vdc
Y1
24.576MHz
MULTIPLE DEVICE DAISY CHAIN PORT
TO AC97 DIGITAL INTERFACE
CHASSI GROUND
ANALOG GROUND
DIGITAL GROUND
FB10 600Z
AD1819 SINGLE MAFE AUDIO I/O
C:\CLIEWORK\LARRY\AD19MAFE.SCH
LETTER
Larry Hurst
ANALOG DEVICES, INC., RSTC
804 WOBURN ST
WILMINGTON, MA 01887-3494
11-Dec-1996
96327-01-04
1 1
A

APPENDIX B - AD1819A Indexed Control Register Reference

Reg Num	Name	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Default
00h	Reset	X	SE4	SE3	SE2	SE1	SE0	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0	0400h
02h	Master Volume	MM	X	LMV5	LMV4	LMV3	LMV2	LMV1	LMV0	X	X	RMV5	RMV4	RMV3	RMV2	RMV1	RMV0	8000h
04h	Reserved	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
06h	Master Volume Mono	MMM	X	X	X	X	X	X	X	X	X	MMV5	MMV4	MMV2	MMV2	MMV1	MMV0	8000h
08h	Reserved	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
0Ah	PC_BEEP Volume	PCM	X	X	X	X	X	X	X	X	X	X	PCV3	PCV2	PCV1	PCV0	X	8000h
0Ch	Phone Volume	PHM	X	X	X	X	X	X	X	X	X	X	PHV4	PHV3	PHV2	PHV1	PHV0	8008h
0Eh	Mic Volume	MCM	X	X	X	X	X	X	X	X	M20	X	MCV4	MCV3	MCV2	MCV1	MCV0	8008h
10h	Line In Volume	LM	X	X	LLV4	LLV3	LLV2	LLV1	LLV0	X	X	X	RLV4	RLV3	RLV2	RLV1	RLV0	8808h
12h	CD Volume	CVM	X	X	LCV4	LCV3	LCV2	LCV1	LCV0	X	X	X	RCV4	RCV3	RCV2	RCV1	RCV0	8808h
14h	Video Volume	VM	X	X	LVV4	LVV3	LVV2	LVV1	LVV0	X	X	X	RVV4	RVV3	RVV2	RVV1	RVV0	8808h
16h	Aux Volume	AM	X	X	LAV4	LAV3	LAV2	LAV1	LAV0	X	X	X	RAV4	RAV3	RAV2	RAV1	RAV0	8808h
18h	PCM Out Vol	OM	X	X	LOV4	LOV3	LOV2	LOV1	LOV0	X	X	X	ROV4	ROV3	ROV2	ROV1	ROV0	8808h
1Ah	Record Select	X	X	X	X	X	LS2	LS1	LS0	X	X	X	X	X	RS2	RS1	RS0	0000h
1Ch	Record Gain	IM	X	X	X	LIM3	LIM2	LIM1	LIM0	X	X	X	X	RIM3	RIM2	RIM1	RIM0	8000h
1Eh	Reserved	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
20h	General Purpose	POP	X	3D	X	X	X	MIX	MS	LPBK	X	X	X	X	X	X	X	0000h
22h	3D Control	X	X	X	X	X	X	X	X	X	X	X	X	DP3	DP2	DP1	DP0	0000h
24h	Reserved	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
26h	Powerdown Cntrl/Stat	X	X	PR5	PR4	PR3	PR2	PR1	PR0	X	X	X	X	REF	ANL	DAC	ADC	000Xh
28h	Reserved	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
5Ah	Vendor Reserved	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
74h	Serial Configuration	SLOT16	REGM2	REGM1	REGM0	DRQEN	DLRQ2	DLRQ1	DLRQ0	X	X	X	X	X	DRRQ2	DRRQ1	DRRQ0	7000h 7X0Xh
76h	Misc Control Bits	DACZ	X	X	X	X	DLSR	X	ALSR	MODEN	SRX10D7	SRX8D7	X	X	DRSR	X	ARSR	0000h
78h	Sample Rate 0	SR015	SR014	SR013	SR012	SR011	SR010	SR09	SR08	SR07	SR06	SR05	SR04	SR03	SR02	SR01	SR00	BB80h
7Ah	Sample Rate 1	SR115	SR114	SR113	SR112	SR111	SR110	SR19	SR18	SR17	SR16	SR15	SR14	SR13	SR12	SR11	SR10	BB80h
7Ch	Vendor ID1	F7	F6	F5	F4	F3	F2	F1	F0	S7	S6	S5	S4	S3	S2	S1	S0	4144h
7Eh	Vendor ID2	T7	T6	T5	T4	T3	T2	T1	T0	REV7	REV6	REV5	REV4	REV3	REV2	REV1	REV0	5300h

AD1819A Block Diagram Register Map





AD1819A Mixer Functional Diagram

Disclaimer

Information furnished in this document by Analog Devices, Inc., is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices Inc., for its use; nor for any infringements of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices. Analog Devices Inc. reserves the right to make changes without further notice to any products and information contained in this document.

Analog Devices makes no guarantee regarding the suitability of its DSP and codec products for any particular purpose, nor does Analog Devices assume any liability arising out of the application or use of Analog Devices DSP and codec products, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. Operating parameters, such as voltage and temperature, as specified in Analog Devices data sheets must be validated for each application by the customer's technical experts. Analog Device's DSP and codec products are not designed, intended, or authorized for use for products in which the failure of the ADI part could create a situation where personal injury or death may occur. If the Buyer/User/Designer uses Analog Devices products for any unintended or unauthorized application, then Analog Devices cannot be held responsible.