

# Application Note **115**

## Using the Audio Codec on ARM Development Boards

Document number: ARM DAI 0115B

Issued: March 2007

Copyright ARM Limited 2007

# ARM

## **Application Note 115**

### **Using the Audio Codec on ARM Development Boards**

Copyright © 2007 ARM Limited. All rights reserved.

#### **Release information**

The following changes have been made to this Application Note.

#### **Change history**

<b>Date</b>	<b>Issue</b>	<b>Change</b>
September 2004	A	First release
March 2007	B	Added explicit support for AB926EJ-S, EB

#### **Proprietary notice**

ARM, the ARM Powered logo, Thumb and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, ARM7TDMI, ARM9TDMI, TDMI and STRONG are trademarks of ARM Limited.

All other products, or services, mentioned herein may be trademarks of their respective owners

#### **Confidentiality status**

This document is Open Access. This document has no restriction on distribution.

#### **Feedback on this Application Note**

If you have any comments on this Application Note, please send email to [errata@arm.com](mailto:errata@arm.com) giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

#### **ARM web address**

<http://www.arm.com>

---

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>4</b>
1.1	ARM Development Board sound subsystem specification .....	4
1.2	Brief Introduction to the PCM audio format .....	5
<b>2</b>	<b>Development Board Audio Hardware Architecture .....</b>	<b>7</b>
2.1	Audio system block diagram .....	7
2.2	Functional description – PL041 PrimeCell AACI .....	8
2.3	The AC'97 'AC Link' protocol .....	9
2.4	Functional description – LM4549 Audio Codec IC .....	10
<b>3</b>	<b>Programming the Audio Subsystem .....</b>	<b>11</b>
3.1	Basic set-up and use of the AACI and Codec .....	11
3.2	Working with different audio sample data widths .....	12
3.3	Working with monophonic audio data .....	13
3.4	Programmable features of the LM4549 codec .....	13
<b>4</b>	<b>AACI Software examples .....</b>	<b>18</b>
4.1	PrimeCell driver software .....	18
4.2	Simple AACI example code .....	18
4.3	DMA Audio example code (DMAC-equipped boards only) .....	19
4.4	Programming an audio clip into flash memory .....	27
4.5	List of files included in the 'Applications Note 115' zip file .....	29

## 1 Introduction

At the release date of this document, the following ARM development boards are equipped with an audio codec:

- Integrator/CP
- Integrator/IM-PD1
- RealView PB926EJ-S
- RealView AB926EJ-S
- RealView Emulation Baseboard (EB)

Note: Since the AB926EJ-S and PB926EJ-S boards are largely similar from a functional perspective, they are often collectively referred to in this Applications Note and accompanying example code as 'xB926EJ-S'.

All these boards have a similar audio subsystem, which allows stereo PCM audio to be recorded or played back at a variety of bit rates. They use the same National Semiconductor LM4549 audio codec IC and the same ARM PrimeCell AACI (Advanced Audio Codec Interface) logic to drive the codec.

The user guides for these boards briefly describe the audio codec IC and the PrimeCell AACI (PL041). However, what may not be clear from the documentation is which parts of each component can and can not be used, how the two work together, and hence what the specifications of the resultant audio sub-system are. The CD-ROMs supplied with the Integrator boards also contain PrimeCell driver source code, but this is quite complex, since it contains code to exercise each part of the PrimeCell; even the areas which are unused on the Integrator. Some customers have encountered difficulty in using all this information together to enable them to write software for this peripheral.

This document attempts to describe how the LM4549 audio codec IC and ARM PL041 AACI PrimeCell function together in the above development systems. Example C source code is also available which demonstrates how to make the audio system perform simple record and playback operations, as well as control some of the features of the codec itself, such as gain and sample rate.

### 1.1 ARM Development Board sound subsystem specification

The following table lists the various audio subsystem parameters:

Parameter	Value / Comments	
Raw digital audio data format	PCM	
Number of audio channels	2 (Stereo)	
Audio sample data width	12, 16 or 18-bit native (18-bit default). Other data sizes require software conversion of sample data.	
Sample rates supported	4kHz to 48kHz (48kHz default). Variable in 1Hz steps. Record and playback sample rates can be independently selected.	
Audio power output	250mWRMS into 32 $\Omega$ (except IM-LT1 which has no power amplifier)	
FIFO depth	According to the user documentation, the standard PrimeCell PL041 AACI has 20bits x 8 levels deep when in the default 'non-Compact mode' (see section 2.2 for explanation). This has been increased on some platforms:	
	Integrator/IM-PD1 + LM combination	32-bit x 16 deep
	Integrator/CP	32-bit x 256 deep
	PB926EJ-S	32-bit x 256 deep
	AB926EJ-S	32-bit x 256 deep
	EB	32-bit x 256 deep

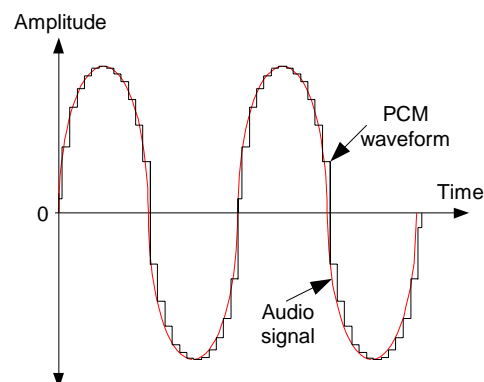
**Table 1-1 Audio subsystem specification**

## 1.2 Brief Introduction to the PCM audio format

There are numerous publications which fully explain this method of digitizing an analogue signal, but here is a brief explanation. Skip this section if you are familiar with the principles of PCM.

PCM means Pulse Code Modulation. An audio signal is represented by a sequence of numbers, each of which corresponds to a DC level at a fixed point in time on an audio waveform. Depending on whether recording or playing back audio, the DC level is either sampled or changed at a particular frequency which is known as the sample rate.

The following diagram shows a sine wave and corresponding PCM waveform.



**Figure 1-1 Example PCM waveform**

### 1.2.1 Bit rate

It is commonly accepted that for a sample rate of  $2f$  Hz, PCM data can only contain information to reconstruct a signal with a maximum frequency of  $f$  Hz. So, for example, a PCM audio recording system with a sample rate of 44.1kHz can capture signals with a maximum frequency of 22.05kHz. Any components with frequencies above this are lost. A lower sample rate corresponds to a reduced audio bandwidth, but the advantage is that fewer samples and consequently less memory are required to store a given audio clip. In the diagram above, the sample rate is approximately 25 times the frequency of the sampled sine wave.

### 1.2.2 Resolution

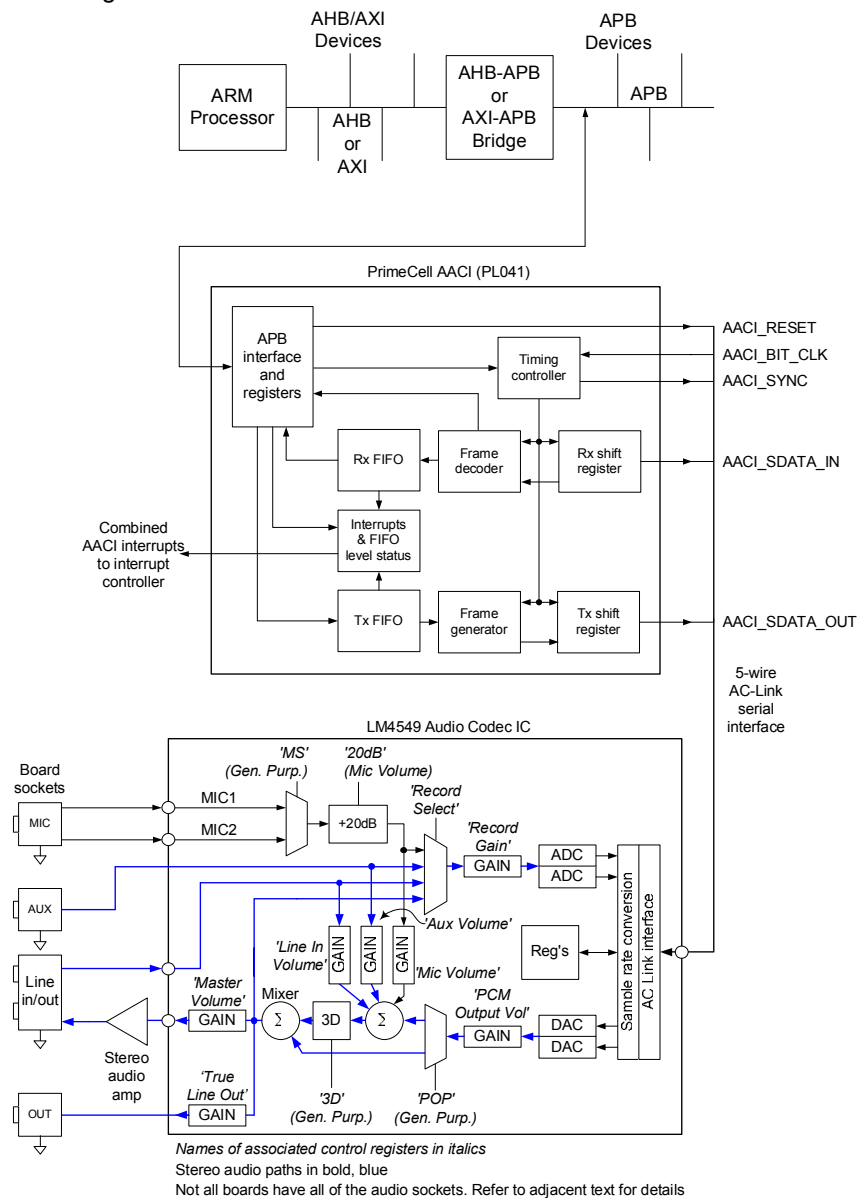
The range of the number system used to capture and store PCM data also has an effect on the quality of the audio signal that can be captured/reconstructed. For example, a 16-bit wide PCM sample (as used on audio CDs) can take one of  $2^{16}$  ( $65536_{10}$ ) values, whereas an 8-bit system can only represent  $2^8$  ( $256_{10}$ ) values. Audio clips captured at lower resolutions obviously require less storage, but the disadvantage is that the sound quality is noticeably poorer. Resolution is also known as 'dynamic range' or 'bits per sample'.

## 2 Development Board Audio Hardware Architecture

This section describes the audio hardware on the aforementioned development systems.

### 2.1 Audio system block diagram

The following block diagram shows how the audio sub-system on the development boards listed above is constructed. The PrimeCell AACI and LM4549 codec blocks have been expanded to show relevant internal detail. Note that the parts of the codec IC which cannot be utilized due to the design of the boards are not shown.



**Figure 2-1 Audio subsystem schematic**

Notes: The IM-PD1 has no amplifier on the line out, and line in/out are on a 5-pin DIN socket. The 3.5mm stereo jacks on the IM-PD1 are connected to 'True Line Out' and 'Aux in' on the LM4549. The following table shows the connector references for each socket on each board type:

Connector	AP	CP	PB	AB	EB
MIC	J7	J12	J4	J3	J2
AUX	J8	-	-	-	-
LINE IN	J29	J10	J3	-	J1
LINE OUT	J29	J10	J3	J2	J1
TRUE LINE OUT	J6	-	-	-	-

**Table 2-1 Audio connector assignments**

## 2.2 Functional description – PL041 PrimeCell AACI

The PrimeCell AACI performs several tasks:

It converts digital audio data stream formats between the raw PCM numeric data handled by the ARM processor, and the AC-Link serial data handled by the codec.

It contains FIFO buffers to ensure that the codec can be supplied with a constant stream of data, and that the data generated by the codec can always be processed by the ARM, without interrupting it too frequently. If the codec is starved of data whilst playing an audio clip, breaks in the sound output will be heard. In the opposite direction, the FIFO ensures that the ARM can store recorded audio data whilst performing other tasks. If the receive FIFO overflows whilst recording, data will be lost.

The audio system can be used in a polled or interrupt-driven manner, although it is likely that all but the lowest throughput systems will use interrupts for audio processing. The AACI can generate interrupts to the ARM, dependant on such things as when the FIFO buffers reach a certain level. There are 15 interrupt sources within the AACI, and these are combined into one interrupt signal, which is fed to the interrupt controller on the development board. It is therefore necessary for the interrupt handler code to interrogate the AACI to determine the exact cause of an AACI interrupt.

The AACI has 4 separate transmit/receive channels, each of which can be configured to handle a number of AC-Link slots. On ARM's development boards however, only one channel is used.

Each channel has transmit and receive FIFOs. The FIFOs can be configured for 'Compact mode' or 'Non-Compact mode'.

In **Non-Compact mode** (the reset default), each 32-bit data word on the AMBA APB (Advanced Peripheral Bus) contains one audio sample. This mode must be used if the maximum (18-bit) resolution is required. The example code that accompanies this Apps Note uses the AACI in Non-Compact mode.

In **Compact mode**, each 32-bit data word on the APB contains two 12 or 16-bit audio samples. The LS half-word is transmitted on the AC-Link interface before the MS half-word, so the LS half-word must contain data for the lowest numbered AC-Link time slot (see section 2.3). In this mode, the LS 16bits are for the left audio channel, and the MS 16bits are for the right audio channel. Compact mode can only be selected when the AACI is set to handle 12 or 16bit data. In Compact mode, the available FIFO depth is halved, since two parallel FIFOs are required to store the L & R audio samples side-by-side. Remember however that only one bus cycle per stereo sample is now required instead of two.



For a full functional description, please refer to the PrimeCell PL041 Technical Reference Manual (TRM).

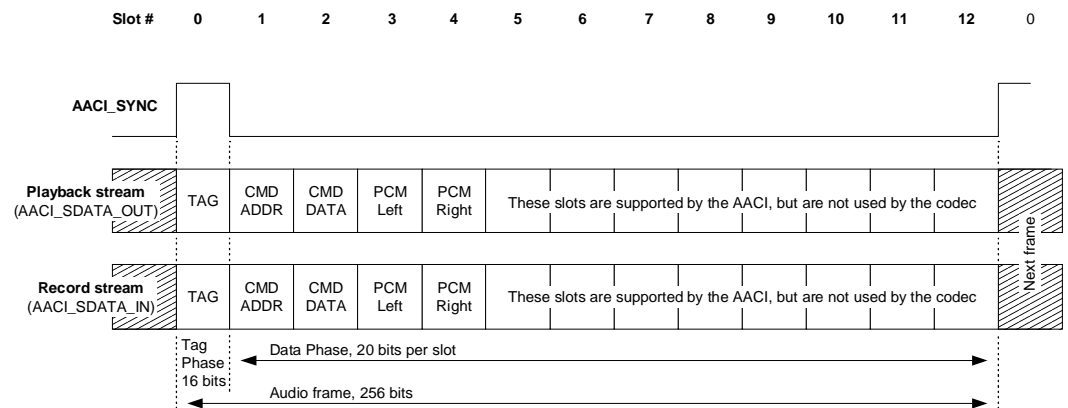
## 2.3 The AC'97 'AC Link' protocol

The AACI TRM describes in detail the 5-wire 'AC-Link' protocol which is used for serial communications between the AACI and codec. This section summarizes parts which are relevant to how this interface is used in ARM development boards.

The AC-Link protocol works on 256-bit audio 'frames', each of which contains 13 time slots. The Codec IC generates a bit clock, AACIBITCLK to time transfers on the interface. The default and maximum bit rate is 12.288MHz, which corresponds to 48,000 audio frames per second, since  $12.288\text{MHz} / 256 = 48\text{kHz}$ . If a sample rate lower than 48kHz is selected by programming the necessary codec registers, the bit clock frequency is reduced accordingly. The AACI outputs the AACI\_SYNC signal, which indicates the start of each frame.

The content of each slot is generated or interpreted automatically by the AACI logic, so the programmer does not necessarily need detailed knowledge of the protocol in order to program the system, but it is useful to have a basic understanding, in order to know how certain aspects of the AACI should be set-up.

The following diagram shows an AC-Link audio frame, as used on ARM development boards. The AACI clock and reset signals are not shown.



**Figure 2-2 Example AC-Link audio data frame**

The first slot (Slot 0) is known as the 'TAG' slot. This contains information about which of the other 12 slots contain valid data. The AACI processes this information dependant on how it has been programmed to handle the subsequent slots.

Slots 1 and 2 contain command data to the codec, and status data from the codec. For example, this could be values and addresses for attenuator and mixer control registers within the codec IC.

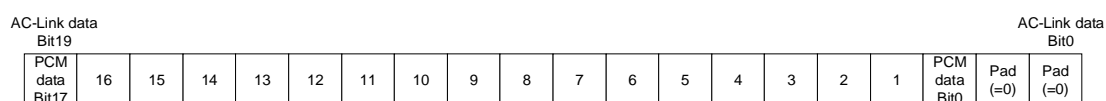
Slots 3 and 4 contain one sample each of left and right audio channel PCM data, to and from the codec.

Slots 5 through 12 are supported by the AACI, but are not used by the LM4549 codec, and consequently are not used on the development boards. In other systems, these could be used for surround sound channels for example.

## 2.4 Functional description – LM4549 Audio Codec IC

The LM4549 audio codec used on ARM's development boards has 18-bit ADCs and DACs. PCM data on this device is represented by 2's complement, signed 18-bit numbers. The LM4549 operates from a single supply rail and as such, does not process bi-polar waveforms (both positive and negative voltages) at its analogue inputs or outputs. The mid or zero point is actually a DC level, approximately half-way between 0 and 5V, but this should be of no concern to the user, since on the board, both the inputs to and outputs from the codec are AC coupled to the 3.5mm jack sockets by capacitors. This means that the audio subsystem cannot be used to produce or measure DC levels.

Although the PCM data is represented by 18-bit quantities, the codec generates and needs to receive 20-bit numbers on the AC-Link serial interface. Within each 20-bit value, the 18-bit PCM sample data is MSB-aligned, and the LSBits are padding. This is shown in the following diagram:



**Figure 2-3 Serial PCM data field**

The codec IC has a number of programmable features, such as a mixer with variable attenuators, Microphone input selector, Sample rate conversion, etc. (See diagram in section 2.1). These features must be accessed by writing to the correct codec register, through the AACI. Please see section 3.4 for more details of codec registers.

---

## 3 Programming the Audio Subsystem

### 3.1 Basic set-up and use of the AACI and Codec

The following algorithms describe the 'bare minimum' set-up required to make the audio subsystem process audio data in the receive and transmit (record and playback) directions. This set-up procedure assumes that all AACI registers are in their default states, as would be the case after a reset. This means that the default 'Non-Compact mode' with 18-bit resolution is used.

Note that the register names listed here and in the example code may have one or more underscore '\_' characters inserted to assist readability. The true register names, as written in the AACI TRM are usually the same, but without the underscores.

- Reset the codec IC. This is done by setting bit0 in the AACI's AACI\_RESET register then clearing it. The value in bit0 of the register is reflected on the AACI\_RESET output pin from the AACI, which resets the Codec. The accompanying demo code generates a reset pulse of 100ms
- To enable audio recording - Program AACI\_RXCR1 register to associate the RX FIFO with the relevant AC-Link slots, namely Slot3 for Right channel data and Slot4 for the Left channel. The receive FIFO is enabled in the same register
- To enable audio playback - Program the AACI\_TXCR1 register to associate the TX FIFO with the relevant audio data slots, and enable transmit FIFO as for receive above
- To enable programming of the Codec IC and operation of the audio subsystem - Program the AACI\_MAINCR register to enable Slot1 (Command address), Slot2 (Command data) and to enable the AACI
- Program Codec volume control, mixer input select, bit rate, etc. as required. This is done by writing the required codec control data and associated register address values to the AACI\_SL2TX and AACI\_SL1TX registers respectively. The registers must be performed in that order, and only when the associated busy flags in the AACI\_SLFR register are clear. Codec reset default settings are: All volume/gain controls at 0dB but muted, record input = MIC1, record and playback sample rates fixed at 48kHz
- Play audio data by repeatedly placing sample values into the transmit FIFO, which is accessed by writing to register AACI\_DR1. This should only be done when the AACI\_TXFF flag in the AACI\_SR1 register is clear, indicating that the FIFO is not full. The AACI will automatically send the samples to the codec at the programmed sample rate
- Record audio data by retrieving sample values from the receive FIFO and storing them in system memory. This is done by reading register AACI\_DR1. The register should only be read when the receive FIFO is not empty; indicated by the AACI\_RXFE flag in the AACI\_SR1 register. Samples will be placed in the FIFO at the selected sample rate.

The simple example code supplied with this Apps Note demonstrates the above procedures.

## 3.2 Working with different audio sample data widths

### 3.2.1 Processing 12, 16, 18 or 20-bit data

The AACI PrimeCell which drives and controls the codec has the ability to process 12, 16, 18 or 20-bit PCM data in hardware. The reset default is 16-bit. The codec IC only works with 18-bit data packed into 20-bit numbers, so the AACI can convert sample data as follows:

#### Recording

The AACI discards some of the least significant bits of the incoming samples by right-shifting the received 20-bit values so that the resultant data is least-significant bit justified.

The number of bits discarded depends on which sample data size is selected in the AACI. This is set by the 'TSize' bits in the AACITXCR1 register. For example, with the AACI set to 16-bit mode, the least significant 4 bits of the 20-bit data words from the codec are discarded, producing 16-bit numbers. This has the effect of reducing the resolution of the recorded audio data from 18-bits to 16-bits.

#### Playback

Again, the AACI can be configured to work with 12, 16, 18 or 20-bit PCM data. The AACI MSB-justifies the PCM data from the ARM into 20-bit numbers, by left-shifting the sample data as required, and padding the least significant bits with zeros. Using the example of 16-bit data again, the AACI would left-shift sample values by 4 bits and insert 4 zero bits of padding before sending the data to the codec.

The highest audio resolution possible on the development system therefore is 18 bits. If 20-bit data is sent to the codec, it will ignore the LS 2 bits. If the AACI is set to receive 20-bit samples from the codec, only 18 bits of resolution will be achieved, since the LS 2 bits from the codec will always be zero.

### 3.2.2 Processing sub 12-bit PCM data

The lowest resolution PCM data that the AACI can process in hardware is 12 bits per sample, so an 8-bit data stream needs some extra work to be done by the ARM processor.

#### Playback

Sample data needs to be MSB-justified into the data size selected in the AACI. For example, if the AACI is set to 16-bit, each 8-bit sample needs to be left-shifted by eight places before being sent to the AACI.

To complicate matters slightly, 8-bit PCM data contained in WAV files is usually stored as unsigned numbers, whereas the LM4549 codec (and most 16-bit WAV files) work with signed quantities. Attempting to play unsigned samples directly will produce a horribly distorted waveform, since any part of the waveform which exceeds the maximum value which can be represented as a 2's complement number will wrap around and appear at the opposite end of the waveform.

Thus, when playing unsigned sample data, an offset which corresponds to the mid-point value must be deducted from each sample value before left-shifting. For 8-bit unsigned values, this offset is 0x7F.

#### Recording

To convert for example, 16-bit samples from the AACI into 8-bit PCM data, the sample values would need to be right-shifted by 8 places, to reduce the resolution to 8 bits. If it is necessary to

---

convert 8-bit signed values into unsigned values, an offset of 0x7F would then need to be added to each sample.

### 3.3 Working with monophonic audio data

There is no hardware setting in the Codec which causes it to process only monophonic audio, but there are several ways to implement mono sound.

Perhaps the most obvious method would be to set-up the AACI to process stereo audio data as normal, but only make use of one analogue audio channel. When recording, the unwanted audio channel data could simply be discarded. On playback, the ARM would have to supply dummy data for the unused channel, so that all FIFO locations were occupied. This method has a problem in that it uses twice as much bus and CPU bandwidth than is actually needed.

A better (and simpler) solution is to only set-up the AACI to process data for one of the two available audio channels. The AACI then no longer interleaves Left and Right samples, but associates each sequential parallel data word with the same audio channel. To do this, only the Left audio channel (AC-Link Slot 3) should be enabled in the AACI\_TXCR1 or AACI\_RXCR1 registers. This method does not work if only the Right channel (Slot 4) is enabled. Note that if sample data with interleaved stereo samples is played with the AACI setup in this way, the resultant mono audio output will contain both L and R channel data, so will appear to be playing at half speed.

### 3.4 Programmable features of the LM4549 codec

The LM4549 codec contains a number of registers which can be programmed via the AC-Link interface, from the AACI PrimeCell. These allow software control of such features as mixer gain settings and ADC/DAC sample rates. This section contains a table which lists only the codec registers that have effect or use on the ARM development boards. Following the table, there is a brief description of the registers, and the bit slices within the registers. For more information please see the data sheet for the codec chip.

The codec registers are accessed by writing the required address and data values into AACI registers AACISL1TX and AACISL2TX respectively. These registers correspond to AC-Link time slots 1 and 2, which are the 'Command Address' and 'Command Data' slots. Transmission must first be enabled by setting the 'S1TxEn' and 'S2TxEn' bits in the AACIMAINCR register. See the AACI TRM for exact details.

The example C source code supplied with this Apps Note shows how to perform these operations. Please see section 4.2 for further details.

Register name	Register offset	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Default
Reset	+0x00	X	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0x0D40
Master volume	+0x02	Mute	X	X	ML4	ML3	ML2	ML1	ML0	X	X	X	MR4	MR3	MR2	MR1	MR0	0x8000
Line Out volume	+0x04	Mute	X	X	ML4	ML3	ML2	ML1	ML0	X	X	X	MR4	MR3	MR2	MR1	MR0	0x8000
Mic volume	+0x0E	Mute	X	X	X	X	X	X	X	X	20dB	X	GN4	GN3	GN2	GN1	GN0	0x8008
Line in volume	+0x10	Mute	X	X	GL4	GL3	GL2	GL1	GL0	X	X	X	GL4	GL3	GL2	GL1	GL0	0x8808
AUX in volume	+0x16	Mute	X	X	GL4	GL3	GL2	GL1	GL0	X	X	X	GL4	GL3	GL2	GL1	GL0	0x8808
PCM out volume	+0x18	Mute	X	X	GL4	GL3	GL2	GL1	GL0	X	X	X	GL4	GL3	GL2	GL1	GL0	0x8808
Record select	+0x1A		X	X	X	X	SL2	SL1	SL0	X	X	X	X	X	SR2	SR1	SR0	0x0000
Record gain	+0x1C	Mute	X	X	X	GL3	GL2	GL1	GL0	X	X	X	X	GR3	GR2	GR1	GR0	0x8000
General purpose	+0x20	POP	X	3D	X	X	X	MIX	MS	LPBK	X	X	X	GR3	GR2	GR1	GR0	0x0000
Powerdown Ctrl/Stat	+0x26	EAPD	PR6	PR5	PR4	PR3	PR2	PR1	PR0	X	X	X	X	REF	ANL	DAC	ADC	0x000X
Extended Audio Ctrl/Status	+0x2A	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	VRA	0x0000
PCM Front DAC rate	+0x2C	SR15	SR14	SR13	SR12	SR11	SR10	SR9	SR8	SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0	0xBB80
PCM ADC rate	+0x32	SR15	SR14	SR13	SR12	SR11	SR10	SR9	SR8	SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0	0xBB80

Table 3-1 Codec register summary

---

### 3.4.1 Codec Register and bit slice descriptions

#### Reset (Offset +0x00)

Write any value into this register to reset other registers to their default values.

#### Master Volume registers (Offset +0x02, +0x04)

0dB to 46.5dB attenuation in 1.5dB steps for L & R channels, mute. Default is both channels at 0dB, muted.

Mute(15)	Mx[4:0]	
0	0x0	0dB attenuation
0	0x1F	46.5dB attenuation
1	X	Mute

**Table 3-2 Master Volume register**

#### Mic Volume (Offset +0x0E)

+12dB gain to 34.5dB attenuation in 1.5dB steps & mute (mono), 20dB boost select. Default is 0dB, muted.

Mute(15)	Gx[4:0]	
0	0x0	12dB gain
0	0x8	0dB
0	0xF	34.5dB attenuation
1	X	Mute

**Table 3-3 Mic, Line In, PCM Out Volume registers**

#### Line in Volume (Offset +0x10)

+12dB gain to 34.5dB attenuation in 1.5dB steps for L & R channels, mute. Bit slices as per Mic Volume register, except for bit 6. Default is 0dB, muted.

#### PCM Out Volume (Offset +0x18)

+12dB gain to 34.5dB attenuation in 1.5dB steps for L & R channels, mute. Bit slices as per Mic Volume register, except for bit 6. Default is 0dB, muted.

#### Record Select (Offset +0x1A)

Selects which of the 8 possible audio sources is fed to the ADCs. Note that it is possible to select a source which is not connected on the development board. Unused sources are grayed out in the table. Default input is 'Mic In' for both L & R channels.

Sx[2:0]	Source
0	Mic
1	CD in - N/C
2	Video in - N/C
3	Aux in (IM-PD1)
4	Line in
5	Stereo Mix
6	Mono Mix - N/C
7	Phone - N/C

**Table 3-4 Record Select register****Record Gain** (Offset +0x1C)

0dB to +22.5dB in 1.5dB steps for L & R channels, & mute. Default is 0dB, muted.

Mute(15)	Gx[2:0]	
0	0xF	22.5dB gain
0	0x0	0dB
1	X	Mute

**Table 3-5 Record Gain register****General Purpose** (Offset +0x20)

Miscellaneous functions, e.g. 3D on/off, 3D bypass, MIC 1/2 select, ADC/DAC loopback.  
Default is all bits set to 0.

Bit	Name	Function	0	1
15	POP	PCM Out Path	Pre-3D	Post-3D
13	3D	3D sound	On	Off
9	MIX	Mono o/p select	Mix	Mic
8	MS	Mic select	Mic1	Mic2
7	LPBK	DAC -> ADC loopback	?	?

**Table 3-6 General Purpose register****Powerdown Ctrl/Status** (Offset +0x26)

Indicates codec readiness, and allows various blocks to be powered down. A '1' in bits 3:0 means that a block is ready.



Bit	Name	Function	
15	EAPD	External amplifier powerdown	WRITE
14	PR6	Not used	
13	PR5	Internal CLK disable	
12	PR4	AC-Link powerdown	
11	PR3	Analog mixer powerdown (VRef off)	
10	PR2	Analog mixer powerdown (VRef on)	
9	PR1	DACs powerdown	
8	PR0	ADCs and input MUX powerdown	
3	REF	Vref's up to nominal level	READ
2	ANL	Analog mixers ready	
1	DAC	DAC section ready to accept data	
0	ADC	ADC section ready to transmit data	

**Table 3-7 Powerdown Ctrl/Status register**

**Extended Audio/Ctrl Status (Offset +0x2A)**

Enables/disables variable bit rate. Default is fixed rate; 48kHz.

Bit	Name	Function	0	1
0	VRA	Variable rate	48kHz	Variable rate

**Table 3-8 Extended Audio/Ctrl Status register**

**PCM DAC Rate (Offset +0x2C)**

Sets DAC bit rate. Default is 48kHz.

**PCM ADC Rate (Offset +0x32)**

Sets ADC bit rate. Default is 48kHz.

## 4 AACI Software examples

This section describes the example source code which is available for the audio subsystem on ARM development boards.

### 4.1 PrimeCell driver software

Some ARM development systems, namely the Integrator/CP and Integrator/IM-PD1 boards were originally supplied with the PrimeCell driver C source code on CD. This software was supplied under a development license, which meant that it could not be used in an application without a specific license agreement from ARM.

The PrimeCell driver code supplied with these ARM development boards is arranged into a menu-driven test program for all of the PrimeCell peripherals on the development board.

For many developers, this is not a good starting point for simply trying to get the audio codec working, since there are several layers of abstraction within the source code, and the program itself uses several levels of buffering for the audio data. The PrimeCell AACI driver code is interrupt driven and the interrupt handler code is large, since it contains code for all the peripheral interrupt sources. These factors can make it difficult to extract the relevant information about how the audio subsystem should be programmed.

With this in mind, we have created some simple demonstration code, which is described below:

### 4.2 Simple AACI example code

Due to the complexity of the PrimeCell driver code, we have provided some very simple, non-interrupt driven code which demonstrates how to set up the AACI and codec IC to perform basic record and playback functions. This code is based on the self-test code shipped with the development boards, and is available for download from the ARM website, along with this Applications Note.

The simple example code demonstrates how to perform the following operations:

- Simple, 'bare minimum' set-up for the AACI and codec
- Software control of codec chip mixer level controls for Master volume, PCM playback volume, Record gain, ADC and DAC rates
- Playback a 16-bit stereo PCM data clip. The clip must have been previously stored in flash memory using the flash programming facilities in the ARM debugger or Boot Monitor. For details on how to do this, see section 4.3.
- Record an audio clip from the MIC1 input into RAM, and then play back that clip
- Play a sine wave tone from a lookup table. Frequency can be changed by adjusting the DAC playback rate.

There is a single set of example source files which can be built to run on one of the platforms listed at the start of section 1 by performing the following steps:

1. Extract the Apps Note demonstration code into a clean directory.
2. Run the batch file which pertains to your development board, e.g. **buildAP.bat**, **buildCP.bat** or **buildVP.bat**. The batch files call the ARM code generation tools with the correct command line options to build the code for your chosen board.

For a full list of files included with this Applications Note, see section 4.5. The code is not built for any particular ARM core type, so the resultant executable is ARM architecture v4 machine code, and does not attempt to make use of cache memories. It makes no use of interrupts, instead polling various status flag registers in the AACI before processing data. One of the timer/counter modules on the chosen platform board are used for timeout and sleep functions, and again this is polled rather than interrupt driven. This arrangement means that when the ARM core is not actually transferring audio data between system memory and the audio subsystem, it is simply spinning cycles, waiting for the timer to time out.

### 4.3 DMA Audio example code (DMAC-equipped boards only)

The following boards contain a PrimeCell DMA Controller, which once initialized can be used to send data to/from the AACI without constant intervention from the CPU:

Board Type	PrimeCell DMAC	DMAC Location
AB926EJ-S	PL080	Dev Chip
PB926EJ-S	PL080	Dev Chip
EB*	PL081	FPGA

**Table 4-1 DMAC PrimeCell part number and physical location**

\* Whether a DMAC is present in an EB-based system depends largely on the version of the FPGA image. The FPGA must be re-programmed for each different type of Core Tile used with the EB. Please refer to the Application Note that relates to your specific EB FPGA build for information on the presence of a DMA Controller.

The following build batch files will produce executable code that contains the DMA demonstration:

- BuildAB926.bat
- BuildPB926.bat
- BuildEB.bat

These builds of the example code demonstrate use of a DMAC to transfer a pre-recorded audio clip from Flash memory to the AACI.

The supplied DMA example code can be run by choosing the relevant item from the AACI demonstration program menu. Only one DMA demonstration is included, which plays a fixed length selection of PCM audio data from the Line Out socket on the development board. The audio data must previously have been stored in Flash memory at the address used by the specific build of the program:

Build	Flash address
xB926EJ-S	0x36000000
EB	0x43800000

**Table 4-2 Flash memory location for pre-recorded audio clip**

If the Flash is not pre-programmed with the audio clip data, the AACI will try to 'play' whatever numerical values are stored in Flash. This might amount to nothing if the Flash is erased, or something approximating white noise if the Flash contains program opcodes. For details of how to program PCM audio data into flash, please see section 4.4.

Note: None of the Integrator systems available 'off-the-shelf' contains a DMA controller, although it would be possible for a developer to add one, were there enough free FPGA resource available in the system. For this reason, this DMA example code does not have build options for Integrator boards.

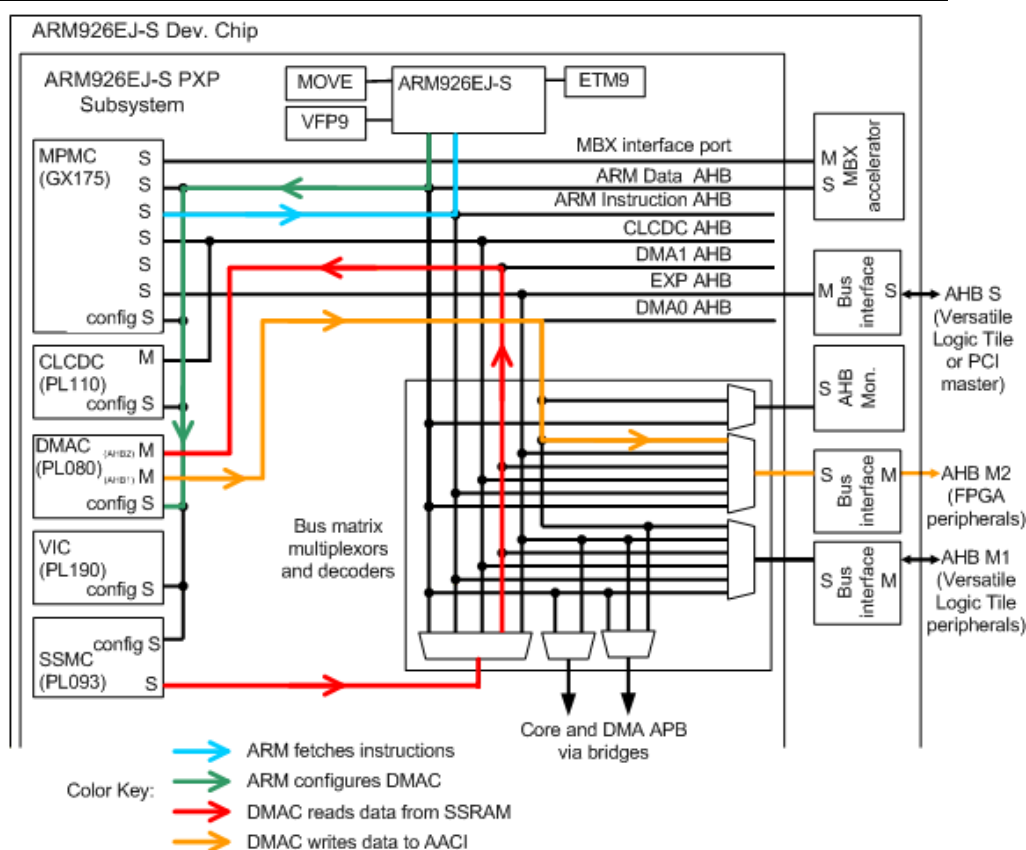
The example code sets up the DMAC to perform memory to peripheral transfers, with the DMAC as the flow controller. This means that the DMAC performs a finite number of preprogrammed transfers. If the DMAC were configured for Memory -> Peripheral transfer with the peripheral as the flow controller, the AACI would never stop playing audio, since it has no way of knowing when the audio data has all been played and would continue to request bursts of data from the DMAC indefinitely.

#### 4.3.1 xB926EJ-S bus architecture and DMA data transfer path

The AB926EJ-S and PB926EJ-S development boards utilize the same 'Development Chip', in which there is an ARM926EJ-S processor core, some peripherals and a 6-layer bus matrix. This maximizes data throughput since bus transactions can take place between devices on different layers of the bus simultaneously. In summary:

1. The ARM926EJ-S core fetches and executes the program instructions from the SDRAM controller on the ARM Instruction-AHB bus layer.
2. The ARM writes configuration data to the DMAC's slave port on the ARM Data-AHB bus layer. (There are also writes to the SSRAM area (SSMC) when the LLI table is populated, but this is not shown for clarity.)
3. Once enabled, the DMAC fetches data from the Flash memory (SSMC) using its AHB1 master port...
4. ...and writes the same data out of its AHB2 master port to the AACI in the board's FPGA. This data exits the Dev. Chip on the external 'M2' bus.

Items 3 and 4 above are repeated until the DMAC has transferred the last block of data to the AACI. This is shown in the following diagram:



**Figure 4-1 xB926EJ-S bus architecture and DMA data path**

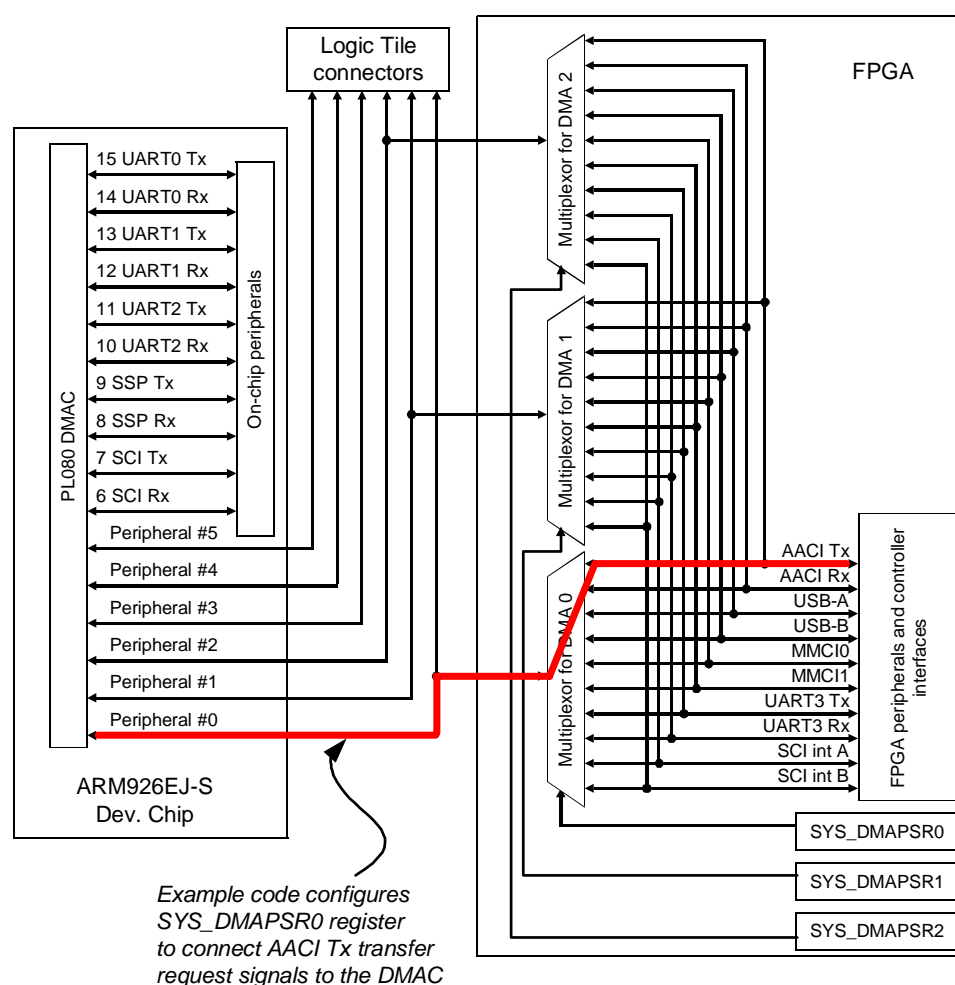
#### 4.3.2 DMA handshake signal routing on the PB926EJ-S

The PL080 DMAC has eight separate ‘channels’ which can be used to configure and control eight separate DMA transfer sequences to/from memory or peripherals. It also has sixteen sets of peripheral handshaking lines, so can be connected to a maximum of sixteen peripheral devices in a system. A ‘channel’ can be programmed to manage a DMA transfer sequence between two specific devices, and then it can be re-programmed for another transfer sequence.

In the PB926EJ-S system there are more than 16 devices that have DMA handshaking signals, and the lowest three handshaking signals (0, 1, 2) into the DMAC are shared between a number of peripheral devices in the PB926 FPGA and up to 3 devices in an attached logic tile. The three DMA signals from the logic tile stack are always connected to DMAC request lines 0, 1, 2. In addition, there are three multiplexers in the PB926 FPGA which can be configured (by the **SYS\_DMAPSRx** registers in the FPGA) to connect any three of the ten devices in the FPGA to the DMAC.

These are: AACI Rx and TX, USB ‘A’ and ‘B’, MMCI 0 and 1. It is important that the logic tile and FPGA do not drive the same DMA request line.

This multiplexing arrangement is shown in the following diagram:



**Figure 4-2 DMA handshake signal MUXing on the PB926EJ-S**

#### 4.3.3 DMA handshake signal routing on the AB926EJ-S

The AB926EJ-S board uses the same Dev Chip as the PB926EJ-S board, and also uses the same PL080 DMAC inside its FPGA. The DMA handshake signal routing differs between the two boards however. Since the AB926EJ-S board has a reduced feature set from the PB926EJ-S board, it is not possible to perform DMA transfers with the same range of peripherals. DMA handshake signal routing is fixed and there are no `SYS_DMAPSRx` registers. The mapping is as follows:

DMA Peripheral #	Peripheral
0	AACI Rx
1	AACI Tx
2	MCIO

**Figure 4-3 DMAC Peripheral numbers on the AB926EJ-S**

Attempts to access the non-existent `SYS_DMAPSRx` registers will not result in any kind of bus error since the AHB-APB bridge returns an OKAY response to all transfers; even if the LSbits of

---

the address do not decode to anything. The example software utilizes this fact to make the code simpler, as the same code can be used for both the AB and PB926 boards.

#### 4.3.4 EB bus architecture and DMA data transfer path

The bus architecture of an EB system is defined by the image programmed into the EB FPGA. The following diagram is from the Applications Note AN152 - *Using a CT11MPCore with the RealView™ Emulation Baseboard*. Either an AXI or AHB bus matrix will be employed, depending on which ARM core type the FPGA image was designed to work with. Please refer to the Applications Note that relates to your system configuration for exact details.

Data flow is very similar to the xB926EJ-S systems, but EB systems use a single-master DMAC.

1. The ARM core fetches and executes the program instructions from the SDRAM controller
2. The ARM writes configuration data to the DMAC's slave port on the ARM Data-AHB bus layer. (There are also writes to the SSRAM area (SSMC) when the LLI table is populated, but this is not shown for clarity.)
3. Once enabled, the DMAC fetches data from the Flash memory (SSMC)...
4. ...and writes the same data out to the AACI.

The following diagram shows the general data flow in an EB system:

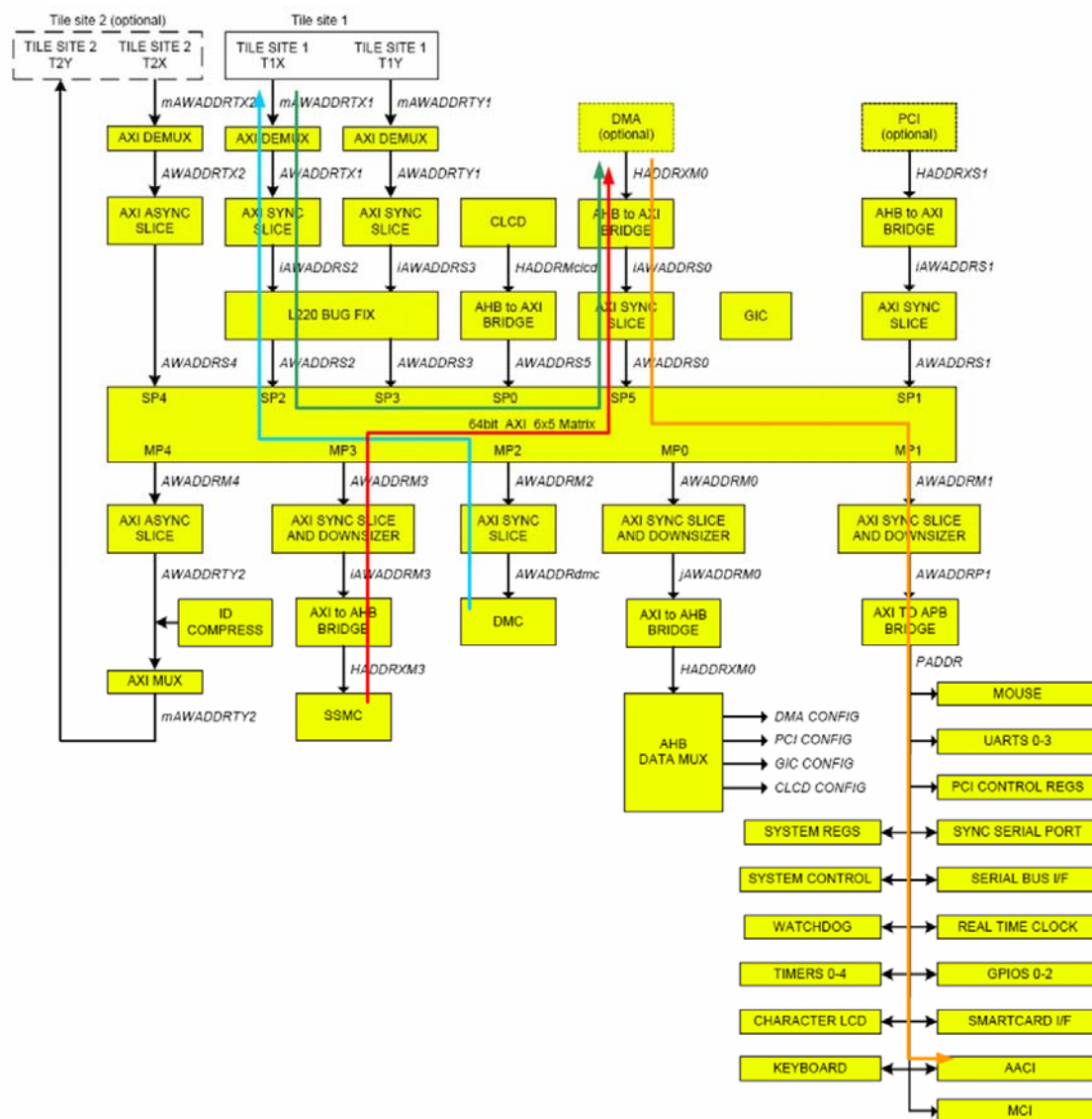


Figure 4-4 EB bus architecture and DMA data path

#### 4.3.5 DMA handshake signal routing on the EB

The EB uses a similar multiplexing system to the PB926EJ-S, with some peripherals connected directly to the DMAC and some being multiplexed with the Logic Tile DMA lines. Fortunately for this application, the AACI DMA lines have exclusive connections to the DMAC in the FPGA. This makes the set-up code simpler.

The following table shows some of the DMAC to Peripheral connections in an EB FPGA image:



---

DMAC Peripheral #	Peripheral
0 – 2	User defined – from Tile sites
3	MCI
4	AACI Rx
5	AACI Tx
[6:15]	Other peripherals - see user guide if interested

**Figure 4-5 DMAC Peripheral numbers on the EB**

#### 4.3.6 Use of linked lists to transfer a large block of data

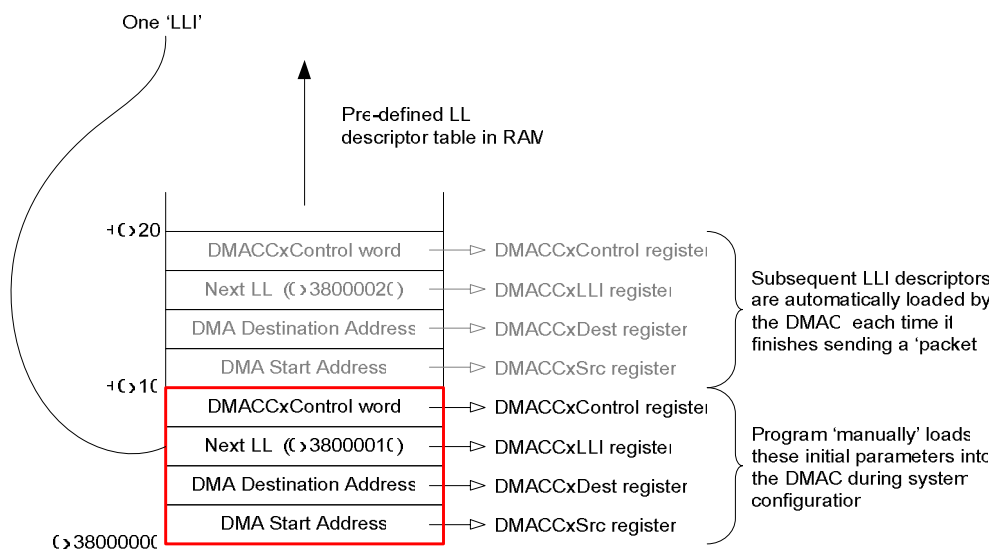
There is a hierarchy to the way in which data is transferred by the DMAC. An entire preprogrammed transfer sequence is split up into one or more DMA ‘packets’. One packet is made up of a maximum of 4095 ‘transfers’ (transactions on the AMBA bus), which can be grouped together in ‘bursts’ of up to 256 transfers. Note that it is not possible to directly specify which kind of AMBA burst (E.g. AHB burst types: INCR, INCR4, INCR8 - as indicated by the HBURST signals) is generated; the DMAC decides which AMBA transfer type to use.

The PL080 and PL081 DMACs used on the xB926EJ-S and EB systems are AHB devices. Where an AXI bus matrix is used in the EB FPGA, an AHB-AXI bridge is employed for protocol conversion (see Figure 4-4).

If more than one packet of data is to be transferred, this can be automated by generating a linked list in memory, which contains a series of 4-word descriptors or LLIs (Linked List Items) that tell the DMAC where to get the data from, where to send it, where to find the next LLI, and other options for transfer of the packet. Once initialized, the DMAC automatically loads LLIs from memory until it loads an LLI which tells it to stop transferring data, or the DMAC is manually disabled.

This code example creates a circular linked list in the baseboard SRAM area before it configures the DMAC. Once enabled, the DMAC follows the linked list, playing a loop of audio until the user instructs the program to disable the DMAC transfer.

Each linked list item (LLI) contains the following data:



**Figure 4-6 Linked List for DMAC control**

Note: The base address shown for the SSRAM area in the diagram is only correct for the xB926EJ-S systems. The EB baseboard SRAM starts at 0x48000000.

The DMACCxControl word contains information which configures the DMAC for the current transfer packet. Parameters are described in full in the source code comments and DMAC TRM, but in summary they are: Number of 'transfers' from the source in the packet (where a 'transfer' can be 8, 16 or 32-bits wide), Number of transfers in each burst for both source and destination, Source and destination transfer widths, DMAC AHB master port to use for source and destination transfers (only relevant for the dual-master DMAC PL080), Whether to automatically increment source and destination addresses after each transfer, the value that should be presented on the AMBA HPROT lines for this packet of transfers, and finally whether a 'Terminal Count' interrupt will be generated at the end of the packet.

In this example, each DMA packet contains 2048 transfers, and each transfer to the AACI is configured to be 16-bits wide, to match the AACI setup in the rest of the example. Transfers from Flash are set to be 32-bits wide, to make efficient use of available bus bandwidth. Therefore, 4KB of data is transferred for each LLI loaded. The final LLI has the 'Next LLI' field set to the address of the first LLI, which makes the DMAC continually transfer the same selection of audio data. 'STOP' can be pressed in the debugger, and the audio should continue to play. The program prompts the user to hit 'Enter' when required, and the ARM then manually stops the DMAC transfer.

Note that if using the default semihosting method in the debugger, the ARM core will be stopped anyway when the program is waiting for the user to hit 'Enter' at the scanf() function.

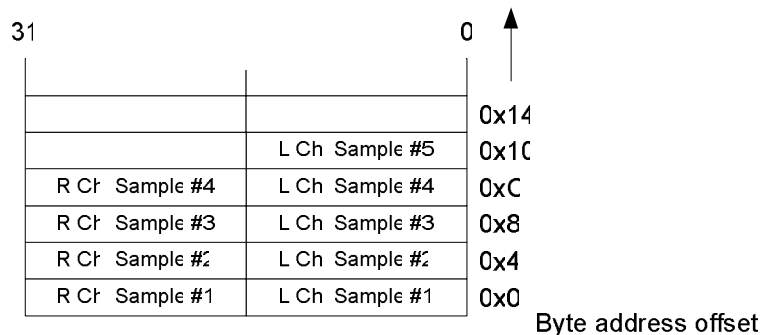
In a typical 'real world' system, the LLI would not loop, but may be set-up so that the final LLI terminates the transfer. This is done by setting the 'Next LLI' field to zero. The DMAC TC (Terminal Count) interrupt could be enabled, and would be routed to the ARM core to indicate that the DMA channel has finished the transfer of data, so could be re-programmed for another transfer; either with the same peripheral, or re-assigned to another peripheral. Note also that a real-world application might have to find an unused DMA channel before configuring it and attempting the transfer. This example assumes that DMA channel 0 will always be available.

---

## 4.4 Programming an audio clip into flash memory

### 4.4.1 Producing suitable audio data

Some of the demo program functions ('Play pre-recorded PCM data from flash memory' and 'Run DMA demo') play an audio clip of 16-bit stereo samples from sequential locations in the development board's flash memory. In order for these to work, the data must be programmed into Flash. First, a binary file must be produced which contains only the raw PCM data, arranged thus:



**Figure 4-7 Required 16-bit PCM data format**

It is relatively easy to create such a file from a 16-bit stereo WAV format file, by editing the WAV file with a hex editor, and stripping away any header information. A WAV file header can be of slightly different lengths depending on the exact variant of the format, but removing the first 0x40 bytes from a WAV file usually suffices. It is also possible to remove areas of blank space (no audio output) from a file by deleting large areas of no signal (sample values at or near zero for signed data). Note that if an odd number of bytes are removed then the sample data will become nonsensical and will sound like noise when played.

An example binary file (telephon.pcm) is included in the supporting files for this Apps Note. It contains the sound of a telephone ringing, sampled at 5.5kHz.

### 4.4.2 Placing the audio data in Flash memory

The binary file must be placed by the user at the point in flash memory which the demo program is set-up to read from. On the Integrator builds of the demo program, this is 0x24000000, which is the start of the 32MB 'Application Flash' area. On the xB926EJ-S boards this is 0x36000000 and on the EB 0x43800000, which is part-way up the 64MB 'NOR Flash' area. This will usually remove the need for the user to erase existing images from flash memory or change the address in the code.

There are several methods for programming flash memory on the ARM development boards:

#### **Integrator (AP, CP):**

- AFU.axf – The ARM Flash Utility is part of the ARM Firmware Suite, which is shipped with Integrator boards. This is a semihosted, menu driven program that allows management of the flash areas on these boards. Use the '?' command to get help for the 'program' command. This program works with the Integrator/AP and CP motherboards.
- Flash.li – This is an AXF file which is downloaded and run on the target system when the File Flash Download menu item is selected in the AXD debugger. There is a GUI front end which controls the operation of the ARM executable. This program only works with the Integrator/AP motherboard.

- Integrator Boot Monitor 'L' command – The boot monitor program installed in Flash on the Integrator/AP and CP motherboards. It is possible to transfer data from a host machine to the Integrator motherboards across an RS232 connection, but this is not recommended since it is very slow. See the user guide for the Integrator system for more information.

**Versatile (xB926EJ-S):**

- RVD's built-in Flash programming functionality – RVD (RealView Debugger) has a built-in Flash programmer, which is invoked whenever the user tries to update the contents of memory which is defined as being Flash memory in a 'board file'. The 'Debug Memory/Register Operations Upload/Download Memory file' menu option can be used to fill an area of memory from a disk file.

**Versatile (EB, xB926EJ-S):**

- V/PB Boot monitor (v1.1 onwards). First, the Flash submenu must be entered by typing 'flash' at the boot monitor prompt, followed by <Return>. Then use the 'write binary' command to place the audio data into Flash. Example command sequence:

```
ARM EB Boot Monitor
Version:      V4.0.5
Build Date:  Dec  7 2006
Tile Site 1:  CT 926EJS
Tile Site 2:  Tile Not Fitted
Endian:       Little
M:\> flash
Flash> write binary c:\temp\telephon.pcm flash_address 0x43800000
Erasing Flash
Writing Flash
Progress 0%
Progress 24%
Progress 50%
Progress 76%
Progress 100%
Flash> list images
Flash Area Base 0x40000000
Address      Name
-----
0x40000000   boot_monitor405LE
Flash Area Base 0x40040000
Address      Name
-----
0x40040000   NFU
0x40080000   u-boot
0x400C0000   uImage
0x40200000   cramfs
0x43800000   telephon
Flash Area Base 0x44000000
Address      Name
-----
Flash Area Base 0x44040000
Address      Name
-----
```

---

Note 1: Using the boot monitor it is possible to program the Flash on the xB926EJ-S boards using Multi-ICE, RealVIEW ICE or the built-in USB debug port. With the EB, only RealView ICE can be used.

Note 2: Later versions of the Versatile Boot Monitor allow an MMC or SD card, mounted as a drive letter to be used as the source for the Flash> write binary and write image commands. Please refer to the baseboard user guide for further information.

- Network Flash Utility (NFU). This utility is available on the CD-ROM that ships with the boards. It must be installed and run from Flash, and allows the user to program Flash memory on the baseboard from files on a TFTP server. The command line interface is very similar to the Boot Monitor 'Flash>' sub-menu and instructions can be found in the baseboard user guides.

#### 4.4.3 Playing MP3 or WAV format files

It is possible to make the development board play WAV files directly (without first stripping and interpreting the header information), provided there are an even number of bytes in the file before the audio data starts. The header information at the start of the file will simply appear as a brief 'pop' from the speakers. To play WAV files properly, code should be included to strip the header information from start of the file, leaving just the PCM data. Code could also be written to interpret the WAV file header and program the AACI and codec accordingly for the correct bit rate, number of channels and resolution.

Other audio file formats, e.g. MP3 could be processed if an appropriate software codec for the ARM processor were obtained. ARM does not currently provide such software.

#### 4.5 List of files included in the 'Applications Note 115' zip file

These are the files included in the demo code zip file:

<b>audiodemo.c</b>	The main Audio demonstration program, containing the C 'main()' function, which provides a rudimentary, 'semihosting'-driven menu system that calls the various demonstration and test routines.
<b>audiodemo.h</b>	Header file which defines some global variables and pointers used by the main() and aaci functions.
<b>aaci.c</b>	Contains functions illustrating how to set-up and control the AACI and Codec, and how to make the system play and record PCM audio data.
<b>aaci.h</b>	Header file containing register addresses and miscellaneous definitions for the PrimeCell PL041 AACI and LM4549 Codec.
<b>dma.c</b>	Contains functions specific to the initialization of the DMA controller.
<b>dma.h</b>	Header file containing register addresses and miscellaneous definitions for the PrimeCell PL080 DMA Controller.
<b>common.c</b>	Sleep/timeout functions with board-specific implementations
<b>common.h</b>	Header file for common.c, containing addresses for timer peripheral etc.
<b>BuildAP.bat</b>	MS-DOS batch file to call the ARM software development tools to build this example code for an Integrator/AP + LM + IM-PD1 system.
<b>BuildCP.bat</b>	Batch file to build the AACI example code for an Integrator/CP system.
<b>BuildAB926.bat</b>	Batch file to build the example code for an AB926EJ-S system.

<b>BuildPB926.bat</b>	Batch file to build the example code for a PB926EJ-S system.
<b>BuildEB.bat</b>	Batch file to build the example code for an EB system.
<b>AN115.pdf</b>	This document.
<b>telephon.pcm</b>	PCM audio file containing the sound of a UK telephone ringing once. Stereo recording, but from a mono source. 5.5kHz sample rate, 16-bit samples.

The source files are well commented, so are not explained in any detail here. The larger header files are cut-down versions of files supplied with the development boards, on CD-ROM. Only the register address and function definitions pertinent to this demo are included in these files, in order to aid understanding. Please refer to the full header files supplied on the board CDs if you need to add extra functionality to this demo code.