KORG

1212 I/O

Low Level Driver Specification

Version 1.0

1.0 SCOPE

This document covers the low level communications that need to exist between the Korg 1212I/O PCI Multi-Channel Audio Interface Card and the application software. This API is written with three points of view. First, the initial applications software to be bundled with the card is MacroMedia's DECK II, which specifies a set of calls in its API. These calls either request information (via <GET> commands) from the PCI card (hereafter referred to as "hardware"), or setup some type of parameter on the hardware (via <SET> commands). Secondly, the DSP code is contained within the low level driver, and is downloaded into the DSP program memory at system startup. The DSP will request a program loading sequence, and the driver will begin loading code via the PCI Mailboxes (more on this later). Lastly, there is memory allocation, card allocation, and all other necessary "Macintosh" low level setup that needs to be done. I hope that the driver programmer will be able to furnish the necessary Macintosh setup commands/standard practices, as I am mainly focused on getting information to and from the hardware.

2.0 Communications Essentials

The hardware contains a PCI device manufactured by PLX Inc., called the PLX9060SD. The 9060 handles all PCI bus chores including bus requests, PCI setups and card IDs, etc. It also provides two primary communication channels: a single, bi-directional DMA channel, and a set of doorbell and mailbox registers.

2.1 DMA

The DMA channel is used to move data from the DSP receive buffers into the host record buffers, and from the host play buffers into the DSP transmit buffers. Of course, this is simplification that I am using just to show how the driver should be thinking of this transfer. The DSP transmit and receive buffers are actually a complex set of memory areas, each with 8 buffers, which must be constructed and deconstructed so that their formats correspond correctly to the FPGA. Also, there is routing and mixing happening before transmitting to the outside world. But, for the driver needs, this simplification will suffice. The following diagram shows this generalized concept:

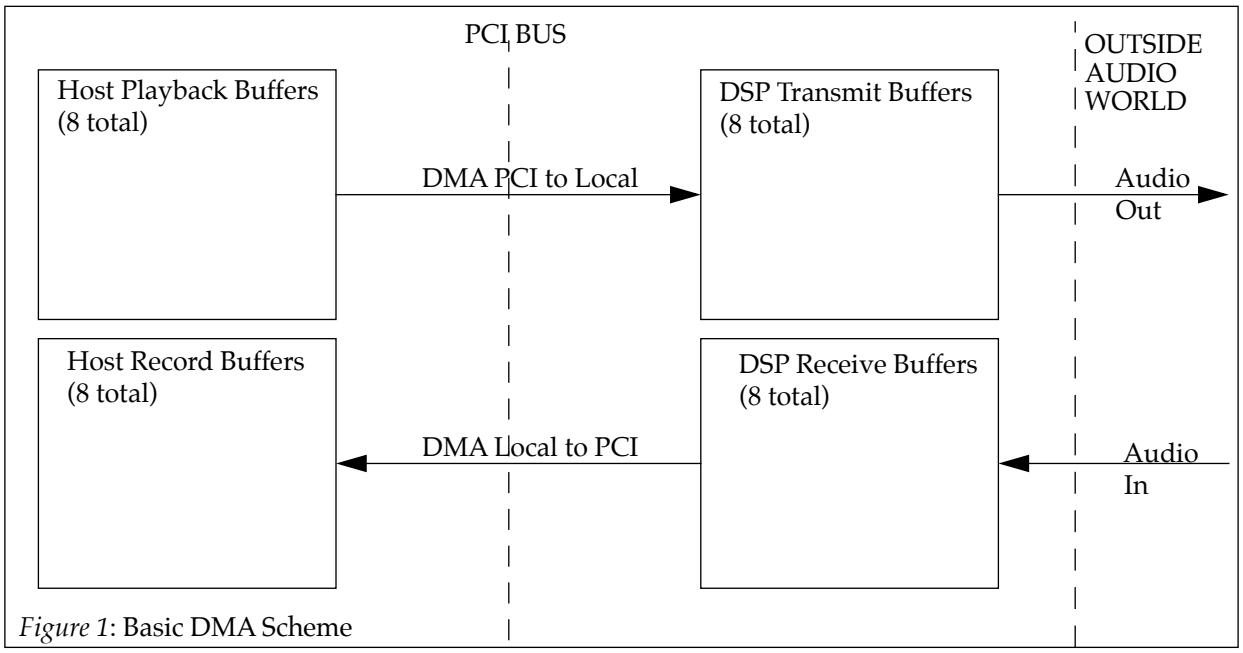

Figure 1: Basic DMA Scheme

The DSP will setup and initiate the DMA transfers. The driver will need to allocate the memory for the host buffers and periodically fill the playback buffers with host data. The DSP will notify the driver that it (the driver) needs to request more audio data every time one of the playback buffers is emptied. The DSP keeps track of this. The application will be constantly emptying the record buffers, and the DSP constantly filling them. This will be discussed in detail later.

2.2 Doorbells and Mailboxes

The 9060 provides two **8-bit** doorbells and four **32-bit** mailboxes. The two doorbells are setup as a PCI-to-Local doorbell and a Local-to-PCI doorbell. When the host needs to tell the hardware something, it rings the PCI-to-Local doorbell, which generates a local interrupt to the DSP. Likewise, when the DSP needs to tell the host something, it rings the Local-to-PCI doorbell, which generates a PCI interrupt. The Driver will need to service this interrupt. Both the host (driver) and DSP (hardware) can read and write to all of the mailbox registers. Therefore, in general, a typical communication via the doorbell/mailboxes would go

something like this (this is a typical configuration example, where the host is configuring the hardware and passing it some info):

- Driver writes some information into one or more of the mailboxes
- Driver rings PCI-to-Local doorbell, setting the doorbell bits according to a pre-defined scheme
- This generates an interrupt to the DSP
- The DSP jumps to its ISR where it then reads the doorbell and clears the set bits (thus clearing the interrupt)
- The DSP decodes the doorbell byte and jumps to a routine for that type of servicing
- The DSP reads the mailbox register(s) and takes the information out
- The DSP returns from its ISR and continues what it was doing

Note that there is no acknowledge at the end of this transfer; the cleared interrupt is really the acknowledge. For a DSP-to-Driver communication, the exact same thing happens, but with the driver reading the doorbell, clearing the interrupt, and then reading the mailbox(es).

It is important to remember that mailboxes are **32-bit** registers. The 9060 breaks this into 4 bytes, which the DSP reads as 2 words. For example, Mailbox 0 is broken into MBox0_LO and MBox0_HI in the DSP-world. This spec will always show which word is important, i.e. the DSP-world designation. In a call, the driver may only need to write a value (e.g. 04h) to the LO word. The spec would say "write the value 0x0004 to **MBox0_LO**." The driver may either write 0x0004 to the MBox0_LO register, or 0x0000 0004 to the entire 32 bit mailbox register. That decision is up to the programmer. In a few cases, the DSP will be expecting 32 bit data in the mailboxes (and must do double reads to access it). In these cases, the spec will say "write the value 0x0004 to **MBox0**" which indicates that all 32 bits must be specified.

2.3 PCI Addresses

Upon bootup, the PCI controller will scan the bus and setup and allocate memory offsets for several PCI address spaces. These include:

- RunTime Base Address (RunTime)
- Address Space 0 Aperture (AS0A)
- PCI Expansion ROM (PER)

These offsets are dynamic, and are held somewhere in the MacOS. Presumably, the driver writer will understand how to find these offsets. Names in parentheses are the offset address names I use in this document. E.G. *RunTime + 060h* would signify 060h spaces from the RunTime Base Address.

2.4 Non-Communication Calls

The application may also request information from the driver in which the driver does not have to talk to the hardware at all. An example of this is the application call, GetPlayChannelCount, in which the driver will ALWAYS return the value 12.

2.5 Important Addresses

Appendix A contains a copy of the registers that will be referenced in this document. Remember to use the PCI addresses and note the directives to Base offsets (see above Section 2.3).

2.6 Conventions

PCI registers are abbreviated as:

Mailboxes = MBox *E.G.* **Mbox0_LO** = low word of Mailbox 0 register, **MBox0** = both words (32 bit data) of Mailbox 0.

3.0 Play and Record Buffer Spaces

The 1212 I/O Driver allocates two multiple-buffer spaces (currently 8 buffers apiece): a Record buffer and Playback buffer. The multiple buffers are concatenated back-to-back, so that the buffer (n+1) begins at the next address after buffer (n) ends.

The buffers are set up as a circular queue. The card starts reading and writing at buffer zero and continues through to the last buffer, at which point it then starts back at buffer zero. The application must keep track of the buffers.

Each Audio Data Frame includes 12 channels worth of audio data, plus ADAT timecode data. The first 10 audio channels are 16 bit data, the last 2 audio channels are 32 bit data (for S/PDIF), and the ADAT timecode is 32 bit data, for a total of 32 bytes (10*2 + 2*4 + 1*4). (The timecode data is only read from the card, and not written to the card; in Playback frames, simply pad this area with zeroes.)

Buffers are expected to be 512 frames long; however, this allocation is dynamic and can change as long as all the buffers are exactly the same length (Length of Record (n) = Record (n+1) = Play (n) = Play (n+1). Accordingly, there is an application call (GetPacketSize) that asks the driver how big these buffers are, and a DSP setup command that gives it the same number. The DSP keeps track of how much data has been read into the buffers and provides a request for more buffer data. Figure 1, below, is an example of one Audio Data Frame as it relates to one of the playback buffers:

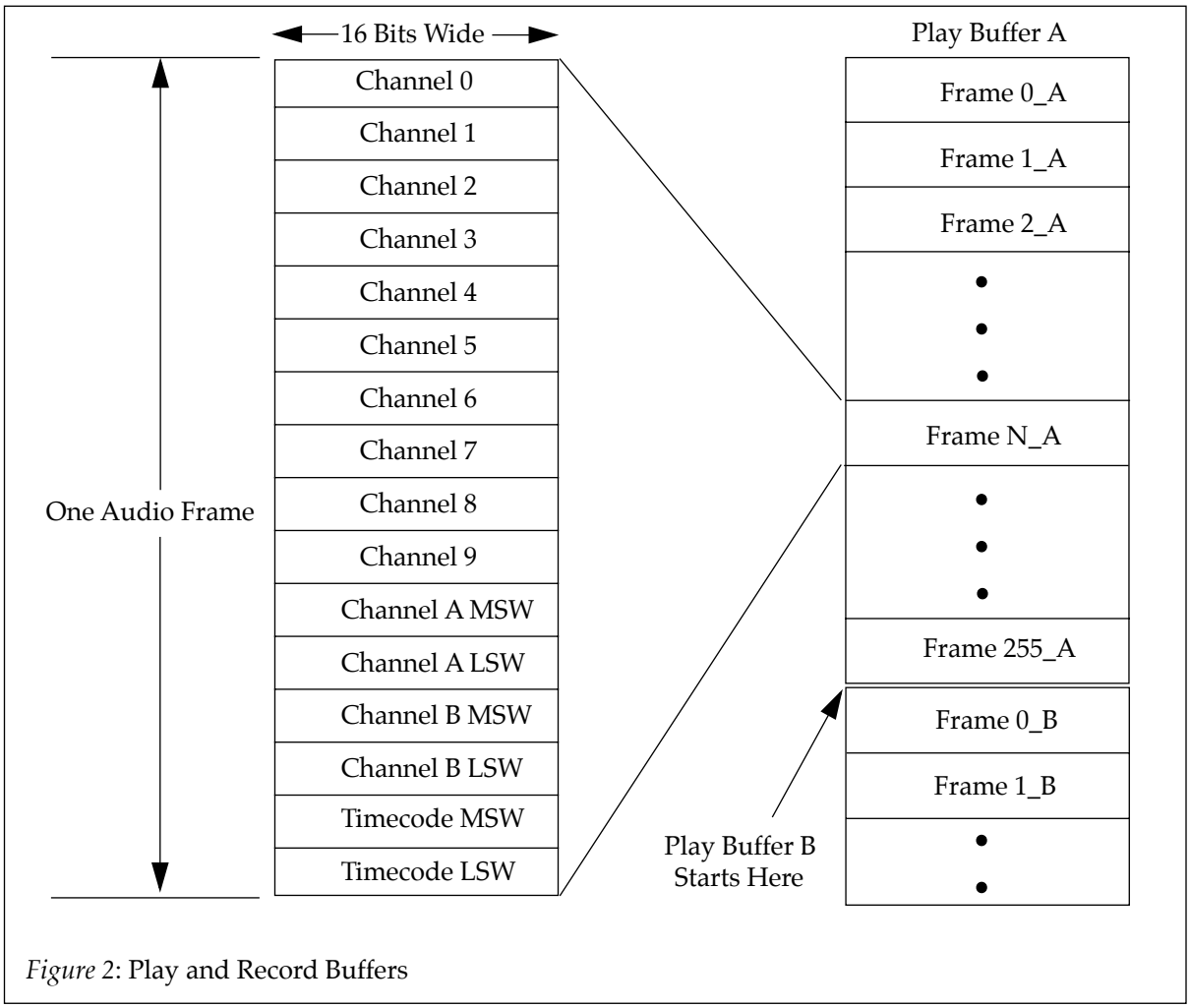

Figure 2: Play and Record Buffers

3.1 S/PDIF Channels

The Korg 1212 I/O has S/PDIF channels located at Channels A and B. Note in the above diagram that these channels are long words and consist of an MSW and LSW channel. This is necessary to support S/PDIF bit depths up to 20 bits. In addition, the 1212 I/O makes the S/PDIF (V)alidity, (C)hannel, and (C)hannel (B)lock (S)tatus bits available to the application. These bits are packed into the S/PDIF long word as follows: (in this example, we are looking at S/PDIF channel A only):

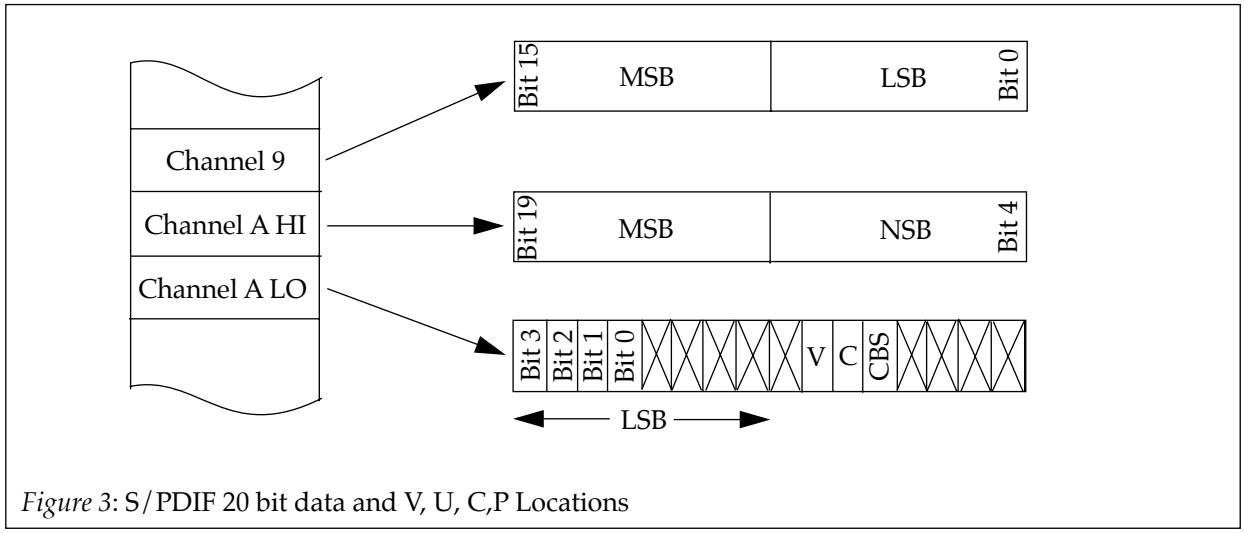

Figure 3: S/PDIF 20 bit data and V, U, C,P Locations

Note the "don't care" bits. The 1212 I/O will *transmit* these as zeros to the application. On the receiving end, the 1212 I/O ignores these bits since the S/PDIF transmitter IC will truncate this byte to 20 bits anyway.

The 1212 I/O transmits the S/PDIF channel bits (V, C, CBS) but ignores the received channel byte. Therefore, these channel bits are read-only and are for information only. The application can not alter these bits in the outbound direction. One instance where these might be useful is in an S/PDIF error-count indicator (much like those on a DAT machine) which monitors the (V)alidity bit and flashes an LED when the received data is invalid. See the S/PDIF professional specification for more details.

For 16-Bit S/PDIF, the application should place this word in the "HI" word slot (Bits 19:4) and fill the LSB with zeros.

4.0 Other Memory Allocations

Besides the Play and Record buffer memory allocation, the driver also needs to allocate spaces for the following data:

4.1 Analog Volume Control Values

The application will make calls to the driver to setup a digitally controlled analog volume IC. There will be both a Left and Right Volume level, that we should NOT assume will always be the same. The driver will receive the volume control change commands and transmit them to the PCI card using a rather tedious method of bit-toggling. The volume control receives 2 bytes per channel, MSB first. The first byte (Byte 0) selects the channel, and will have a value of either 0000 0000b (for Channel 1 or Left) or 0000 0001b (for Channel 2 or Right).

The second byte (Byte 1) will contain the actual volume control information. The driver needs to allocate memory locations (either 2 bytes, 1 word, 1/2 LWord, etc...) to store the last volume code written for each channel. The driver may periodically ask for the last control word written. It is up to the driver programmer to decide how to allocate the memory.

4.2 Channel Volume Control Values

The driver also needs to allocate memory locations for Channel Volume values. There are twelve 16-bit values to be stored, one for each channel in the DSP core. Both the application and the DSP will need to know the starting address of this memory block. See Figure 3 below.

4.3 Channel Routing Control Values

In addition to channel volumes, the driver needs to allocate Channel Routing control values in another contiguous chunk of memory. There are twelve 16-bit values to be stored, one for each channel in the DSP core. Both the application and the DSP will need to know the starting address of this memory block. See figure 3 below.

4.4 State Variable

The driver needs to know its state as either “stopped” or “not stopped.” A state variable should be allocated for this and provisions made. More states may be added later as things finalize.

4.5 ADAT TimeCode Register

The Driver needs to allocate a 2 words of memory for the ADAT Timecode information. The timeCode value is a 32-bit number. We need to discuss how this will be mapped (as bytes/words/LWords).

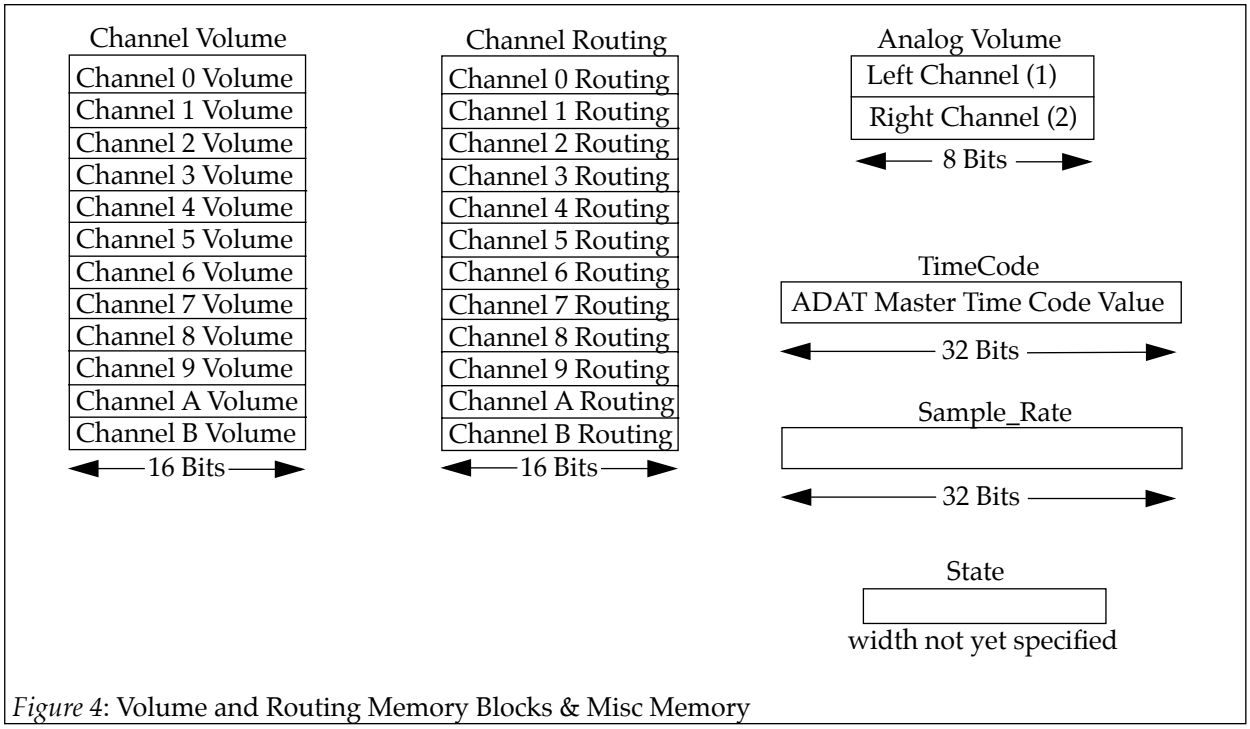

Figure 4: Volume and Routing Memory Blocks & Misc Memory

4.5.1 ADAT Timecode Format

It is beyond the scope of this document to get too detailed about exact ADAT timecode peculiarities. The Alesis document *Alesis ADAT Proprietary Synchronization Interface* (contact Alesis Corp. for more information) contains very detailed information concerning the sync and timecode functions of the ADAT and ADAT XT.

The Korg 1212 I/O reads and generates ADAT timecode. Every four frames, the DMA sequence will result in a freshly updated timecode value. This value is a 32 bit number which represents ***the number of sample clocks that have passed since the very beginning of the tape***. This value must therefore be decoded by the application to produce an hour:minute:second:tenth (00:00:00:00) time value. There are several important considerations when looking at ADAT timecode:

- ADAT timecode can only be read when the ADAT is the clocking master (a limitation due to the AUSY2 ADAT interface chip)
- The actual value (in hours:minutes...) must be calculated by dividing the timecode number by the **sample rate as selected on the face of the ADAT -- this can be 44.1 or 48 KHz.** Since the application knows (and must provide) the clocking information, this should not be a problem.
- In the ADAT itself, vari-speeding the tape will not alter the timecode, but will alter the number of sample clocks. *This creates a severe anomaly since ADAT timecode is defined as "the number of word-clocks that have passed since the beginning of the tape*." The Korg 1212 I/O counts "real" sample clocks (the ADAT counts either 44.1 or 48K clocks, regardless of the true sample frequency) -- as the user vari-speeds the tape, the Korg1212 I/O ADAT timecode value ***will*** slow down or speed up accordingly. This is a trade-off made due to an inefficient timecode reader inside the AUSY2 chip which only allows true timecode reading every 2 sample frames. This is too much overhead for the DSP, so it reads "true" timecode while it chase-locks the ADAT code, then begins generating its own timecode (by counting sample clocks) thereafter.
- The ADAT tape format puts 00:02:00:00 (2 minutes) of a special formatting at the head of each tape. The Timecode is the number of samples since 00:00:00:00, so the first valid, legal timecode that can be produced is: 0x0057 E400 which represents 2 minutes. The math involved is:

57 E400h = 5,760,000d

5,760,000/48000 = 120

120/60 = 2.0 (if the result of this division is greater than 60.0, then another divide-by-60 must be done; these successive divide-by-60s will eventually produce the full, hours:minutes:seconds:tenths number.

4.5.2 Fast ADAT Stop Mode

The 1212I/O includes a DSP switch that allows a mode of operation called "Fast ADAT Stop." The Alesis ADAT timecode specs that a new timecode base will appear every 33 to 65,535 WordClocks (i.e. Fs).

When synchronized to an ADAT, the Application tells the driver the Starting ADAT timecode (a raw, unsigned 32 bit integer). The 1212I/O will chase lock the ADAT, but will wait until the start time is hit before going into play-mode. When the ADAT is stopped, the 1212I/O will also stop.

In order to hit the worst case of 65,535 clocks, the 1212I/O must wait for (65,535 * 20 uSec) or about 1.5 seconds before it can guarantee that the locally generated timecode has "slipped" by too much; the indication that the tape has stopped.

However, for ADATs and ADAT XTs, the timecode base is constantly sent every 960 sample clocks. The 1212I/O has a special function to allow the card to stop playback faster (960 * 20uSec) or 19mSec. This is barely perceptible.

The driver must notify the card ***before*** issuing a play command, that the card will be Syncing to ADAT. This call uses an extra data element to set the Fast ADAT Stop mode. See the API call ***TriggerFromAda***t for more details.

4.6 Record/Playback Latency Issues

4.6.1 Play-Only Latency

In a Playback session, the 1212 I/O first preloads its local buffers. These are analogous to the Host Playback and Record Buffers. After filling its own buffers, the 1212 I/O card updates its host memory pointers. These pointers tell the DSP where, within the Playback and Record buffers, it will need to setup the DMA starting addresses. After that, the DSP waits for a "trigger" signal (a call made through the driver) after which it begins playback. Because it has already preloaded its buffers, and updated its host address pointers, the playback starts immediately, taking two (2) sample periods. Due to the way data I/O is set up within the DSP architecture itself, these 2 cycles are required. Therefore, there is a 2 sample-period latency between the time the trigger is received, and the time that the first playback sample is output.

4.6.2 Record/Play Latency

The second type of latency is the record/playback latency. the 1212 I/O receives playback data frames *from* the host, and sends record frames *to* the host. These events should be sample accurate and locked. In other words, on the same sample period that playback frame N is sent out, the host should receive record frame N. However, the DSP pre-loading scheme, which makes sample playback accurate to 2 sample periods, causes the record data to be offset from the playback data. In fact, this latency is exactly equal to the size of the local DSP sample buffers. Currently, this is 16 sample periods. When the DSP preloads its first buffer full of playback data, it *can not* preload the first buffer of record data to the host, since recording has not yet started. So, it fills the host record buffer with zeros, filling the record buffer with exactly the same number of frames-of-zeros as playback-frames it has preloaded.

For example, say that a play/record session has been initiated. The host has prefilled its Playback buffers with data, and the DSP has prefilled its local buffers with data, and incremented the host address pointer.

At the time that playback frame 0 is output, the 1212 I/O has also acquired record frame 0. However, the latency due to the DSP prefilling routine causes record frame 0 to be placed at the 16th slot in the Host Record Buffer. Examine Figure 4 below:

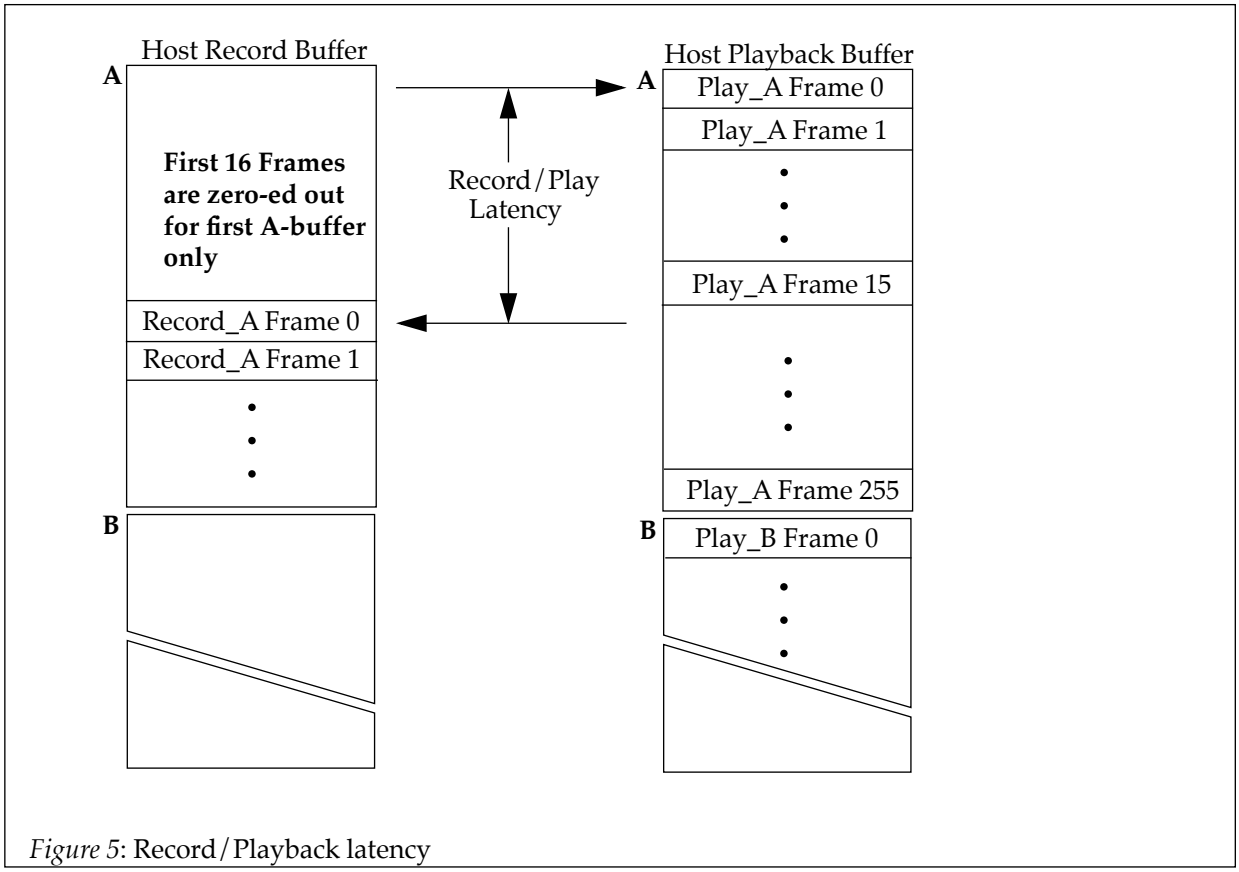

Figure 5: Record/Playback latency

The application should adjust its recorded data storage requirements accordingly so that record frame 0 and play frame 0 "line up" in time.

This latency is related to the size of the local buffers. This buffer sizing is *not dynamically changeable nor is it user-selectable*. However, the buffer size may change as the software (both DSP and driver) are updated. Therefore, the application should make a call at startup requesting the Record/Play Latency time in samples. There is an API call for this very function.

5.0 DSP ucode Download

The DSP code is stored in and shipped with the Korg 1212 I/O driver. When the system boots, or when the Application first calls the driver, the DSP code must be downloaded via the PCI bus. The driver will need to allocate at most 8K bytes of space for the DSP code. The complete board boot sequence is shown below:

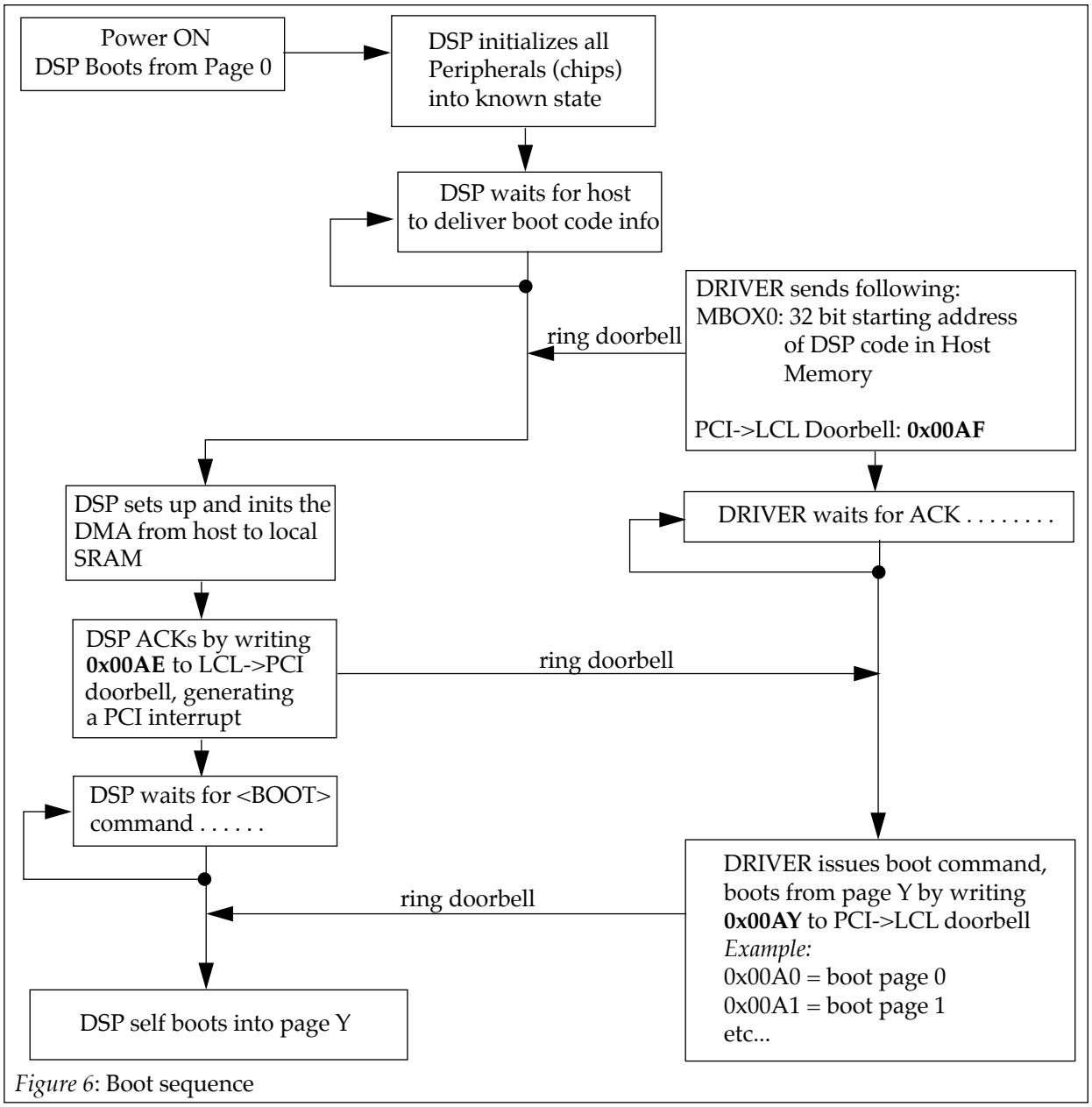

Figure 6: Boot sequence

5.0.1 Boot Notes

After booting, the 1212I/O will need to use the SRAM space that contains the boot code. Therefore, part of the code is written over, and the DSP may never boot again using this code. Therefore, the driver may not issue a boot-from-page-4 command. The driver must issue a boot-from-page-0 command, give the DSP the starting address (of the uCode), then the DSP will do the rest. Although there is a limitation for Boot Page 4 (the SRAM image), there are no limitations on reboots from Pages 0-3 -- these may be done at any time.

6.0 Driver-to-Hardware Communications

These are the calls that will need to be made to the hardware.

These calls to and from hardware are established with the following header:

API Name: This is just a label for this document; may be renamed by programmer
Direction: PCI-to-Local is a call issued by the driver to the hardware
Local-to-PCI is issued by the hardware to the driver via PCI Interrupt
Application-to-Driver is called only to driver; no calls to the hardware
Driver-to-Application is called only to the application; no calls to hardware
Com Type: Either Mailbox, DMA, or Host (Host is for application-to-driver calls)
OTF?: Can this be done On The Fly during playback/record?
Var I/O: lists the variables to be passed and direction (in = into driver; out = out of driver) for Application to Driver or Driver to Application
State: in a few calls, the driver will need to know its state and a variable

6.1 Calls Made TO the Hardware

API Name: SetClockSourceRate
Direction: PCI to Local
Com Type: Mailbox
OTF?: No
Var I/O: in: *long* srate

This call selects the clock source for the card. The hardware can be run from 6 different clocks/rates:

- ADAT Clock @ 44.1KHz
- ADAT Clock @ 48KHz
- S/PDIF Clock or Word Clock In @ 44.1KHz
- S/PDIF Clock or Word Clock In @ 48 KHz
- Local Oscillator @ 44.1 KHz
- Local Oscillator @ 48 KHz

The user has 6 options for setting the sample rate and clock source. The driver will issue this command and pass in the parameter for sample rate/source. The Application may use Table 1 to set this up. This should only be done occasionally. The encoding is:

PROCEDURE:

- Driver receives *srate*
- Driver saves this value in a memory location *sample_rate*
- Load **MBox0_LO** with the corresponding value from Table 1
- Write **0x05** into PCI-to-Local Doorbell

Table 1: Clock Source/Rate vs. Mailbox Value

Clock Source/Rate	MBox0_LO Value
ADAT Input @44.1 KHz	0x8000
ADAT Input @ 48 KHz	0x0000
S/PDIF or Word Clock Input @ 44.1 KHz	0x8001
S/PDIF or Word Clock Input @ 48 KHz	0x0001
Local @ 44.1KHz	0x8002
Local @ 48KHz	0x0002

API Name: ConfigureBufferMemory
Direction: PCI to Local
Com Type: Mailbox
OTF?: Yes

This procedure is called to the hardware after the PCI initialization is complete and the play/record buffer space has been allocated. At this point, the driver will know the starting addresses of the play/record buffers in host memory as well as the size of the buffers (should be 256 frames = 256*28Bytes = 7.168KBytes per half-buffer; one buffer = A half and B half concatenated). The DSP needs to know these starting addresses so that it can set up the DMA blocks correctly. Also, it needs to know the length so it can issue requests to the driver, notifying it that the buffer is half empty.

PROCEDURE:

- **MBox0** = Playback Buffer starting address (32 bit address)
- **MBox1** = Record Buffer starting address
- **MBox2_LO** = length of one buffer (should be 256, but may vary -- see Section 3.0)
- Write **0x03** to PCI-to-Local Doorbell, which initiates the process

API Name: ConfigureMiscMemory
Direction: PCI to Local
Com Type: Mailbox
OTF?: Yes

This procedure is called to the hardware after the PCI initialization is complete and the play/record buffer space has been allocated. The Driver needs to give the DSP the starting addresses of the memory chunks allocated for Channel Volume and Channel Routing arrays as well as the ADAT Timecode address.

PROCEDURE:

- **MBox0** = Channel Volume Control Starting Address (32 Bits)
- **MBox1** = Channel Routing Control Starting Address (32 Bits)
- **MBox2** = ADAT TimeCode Address (32 Bits)
- Write **0x06** to PCI-to-Local Doorbell, which initiates the process

API Name: MonitorOnOff
Direction: PCI to Local
Com Type: Mailbox
OTF?: No
Var I/O: in: *short* newmode (boolean)

This command turns monitor mode on or off. During Monitor mode, there are no DMAs to and from the hardware. In monitor mode, the inputs are looped directly back out to the outputs.

PROCEDURE:

- Driver receives call from application and is given *newmode* = true or false (true=monitor on)
- Driver makes decision and loads MBox0_LO accordingly
- **MBox0_LO** = 0x0002 if Monitor = ON
- **MBox0_LO** = 0x0004 if Monitor = OFF
- Write **0x02** to PCI-to-Local Doorbell

API Name: SetupForPlay
Direction: PCI to Local
Com Type: Mailbox
OTF?: No
Var I/O: no variable passed

This routine is called after the playback buffers have been pre-initialized with data, and record buffers have been zeroed out. This buffer initialization must be done each and every time a play-session is desired. This call will put the PCI card into a play loop where it will initialize its play buffers (by doing the very first DMA) and then it will wait for the trigger.

PROCEDURE:

- Driver receives call for setup. It issues the following:
- **MBox0_LO** = 0x0001
- Write **0x02** to PCI-to-Local Doorbell

API Name: TriggerFromAdat
Direction: PCI to Local
Com Type: Mailbox
OTF?: No
Var I/O: adat_timecode

This sets up the DSP to trigger off of an ADAT timecode instead of the "trigger_play" command below. The application will still issue a "trigger_play" command, but the DSP will wait until the correct ADAT timecode value rolls around before actually triggering.

PROCEDURE:

- Driver receives call for trigger_from_adat, then:
- **MBox0** = adat_timecode (32 bit adat-timecode formatted value)
- **MB0x1_LO**=FastAdatStop mode flag: 0x8000 = fast stop mode, 0x0000 = non-fast-stop mode
- Write **0x07** to PCI-to-Local Doorbell

API Name: TriggerPlay
Direction: PCI to Local
Com Type: Mailbox
OTF?: No
Var I/O: no variable passed
State: save "not stopped" state

This will trigger the playback/record routine in DSP. This can only be called after the Setup_for_Play has been called. The driver will need to toggle its state variable to show that it is "not stopped" -- it should never trigger play if it is not stopped.

PROCEDURE:

- Driver receives call for trigger and immediately:
- Write **0x01** to PCI-to-Local Doorbell
- save state as "not stopped"
- That is all.

API Name: StopPlay
Direction: PCI to Local
Com Type: Mailbox
OTF?: Yes
Var I/O: no variable passed
State: save "stopped" state

This will stop playback or stop monitor mode. The driver should know that it is in the "stop" state by writing to the driver state variable.

PROCEDURE:

- **MBox0_LO** = 0x0008
- Write **0x02** to PCI-to-Local Doorbell

API Name: SetInputSensitivity
Direction: PCI to Local
Com Type: Mailbox
OTF?: No
Var I/O: in: *short* level_1, in: *short* level_2

This sets the record level sensitivity (not really gain, but sensitivity). The input sensitivity controls the ceiling or clip point of the ADC stage. A high sensitivity means "wide-open" or "full-on" while low sensitivity implies "attenuation" or "cutting down the signal." This distinction is made because the 1212 I/O has no pre-amps and thus no gain. A value of 0 is no attenuation (full gain), 0x00FF is maximum attenuation (muted).

This sensitivity setting is accomplished across a 3-wire interface from the PCI chip to a digitally controlled analog volume control. This was added on to the design late in the game. Therefore, the implementation is slightly more complex. The application will pass in the left and right *level* variables and the driver will implement.

Each bit on the 3-wire interface is toggled by writing to a control register on the PLX chip. This 3-wire interface consists of data, clock and load. The load acts as a chip select and load/shift command. The volume control clock can run at a rate up to 1 MHz. Since the driver is toggling the bits, it needs to insert wait loops between clock toggles (noted below as *<wait>*). The programmer should calculate the wait loop to time give about 0.5 usec or more, thus giving a 1 MHz clock rate. The volume control easily runs at lower rates, so anything up to 1 MHz can be used.

The volume control expects data as follows:

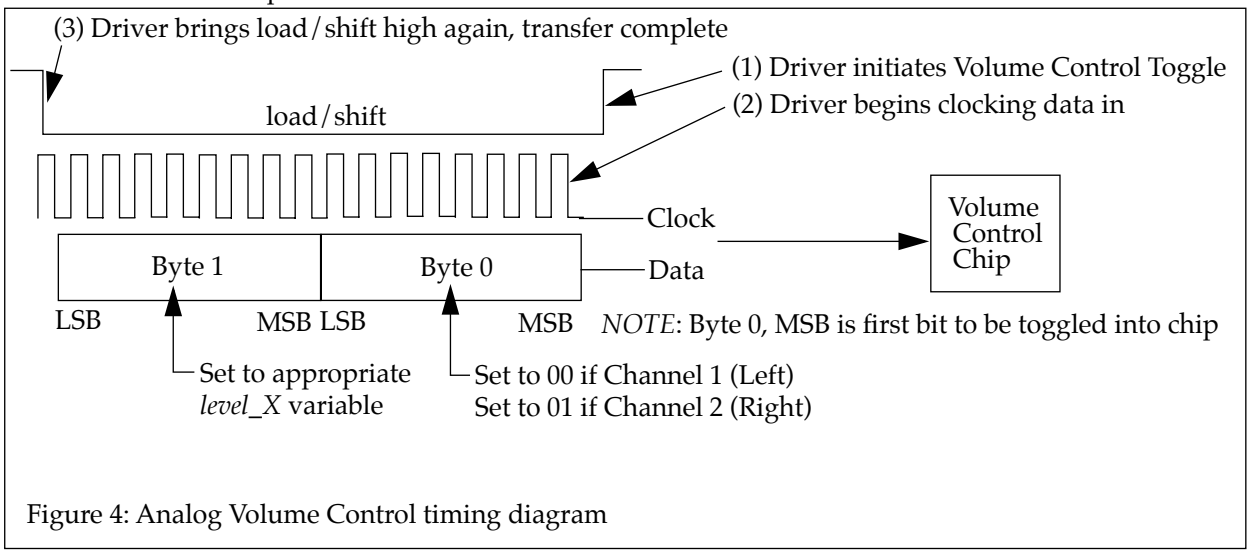

Figure 4: Analog Volume Control timing diagram

PROCEDURE:

- Driver gets call and is passed the *level_1* and *level_2* variables
- Driver writes these values into its Analog Volume memory spaces
- Driver checks the host variable *sample_rate* to determine if 44.1 KHz or 48 KHz (used later)
- Toggle volume control bits by writing to (*PCI RunTime Offset + 06E)* as follows:
- Initiation phase (first data bit, and load/shift toggle):
  - If first data bit = 0: toggle load/shift AND write the first data bit by writing 0x8000,
  - If first data bit = 1: toggle load/shift AND write the first data bit by writing 0x8400,
- Clocking Data Phase (writing all remaining data) -- clock bit must be toggled twice per data bit
  - If next data bit = 0: write 0x8000, *<wait>*, write 0x8100 ,*<wait>*
  - If next data bit = 1: write 0x84000, *<wait>*, write 0x8500, *<wait>*
  - repeat for all data bits
- EOT Phase: transfer complete
  - toggle load/shift again by writing 0x8001
- S/PDIF FF Phase: Need to re-write the SPDIF FF
  - Driver had determined if 44.1 KHz or 48K Hz above
  - If 44.1 KHz, write 0x8001 to (*PCI RunTime Offset + 06E), then*
    - write 0x8101 followed by 0x8001
  - If 48 KHz, write 0x8401 to (*PCI RunTime Offset + 06E), then*
    - write 0x8501 followed by 0x8401

6.2 Calls to the Driver only (no interaction with the hardware)

API Name: **GetClockSourceRate**
Direction: **PCI to Local**
Com Type: **Mailbox**
OTF?: **No**
Var I/O: **out: *long* *srate**

The application calls this to know the address of the *sample_rate* variable. The driver should return this address (pointer). See also Set_Clock_Source_Rate.

API Name: **GetPacketSize**
Direction: **Application to Driver**
Com Type: **Host**
OTF?: **No**
Var I/O: **out: *short* *psize**

The driver needs to allocate a memory location that will hold the value of the packet-size. A packet is really one buffer-of-frames. This should be 512. Upon boot and setup, the driver will load the buffer-of-frames size (hopefully 512) into the memory location *psize*. If the Get_Packet_Size call is made, the driver returns the size (512).

API Name: **GetSampleSize**
Direction: **Application to Driver**
Com Type: **Host**
OTF?: **No**
Var I/O: **out: *short* *sample_size**

The driver will setup a memory location called *sample_size* and fill with the number 2 (2 bytes per sample). This call returns that value.

API Name: **GetPlayCount**
Direction: **Application to Driver**
Com Type: **Host**
OTF?: **No**
Var I/O: **out: *short* *count**

The driver will setup a memory location called *count*, and fill with the number 12. This call returns the value 12.

API Name: **GetRecordCount**
Direction: **Application to Driver**
Com Type: **Host**
OTF?: **No**
Var I/O: **out: *short* *count**

The driver will setup a memory location called *count*, and fill with the number 12. This call returns the value 12.

API Name: GetInputSensitivity
DECK Call: GetInputGain
Direction: Application to Driver
Com Type: Host
OTF?: No
Var I/O: out: *short* *level

The driver allocates a memory location which holds the value *level;* the analog sensitivity level. This call returns the address of (pointer to) the *level* memory location.

API Name: GetPlayBufferAddresses
Direction: Application to Driver
Com Type: Host
OTF?: No
Var I/O: out: *char* **b1, *char* ** b2

This is an initialization call from the application that will ask for the addresses of the Play buffers. These addresses are the starting addresses of the A and B buffers, which are concatenated back-to-back. See *Figure 2: Play and Record Buffers* for a conceptual drawing. The driver then returns the pointer to the pointer of the start addresses.

API Name: GetRecordBufferAddresses
Direction: Application to Driver
Com Type: Host
OTF?: No
Var I/O: out: *char* **b1, *char* ** b2

This is an initialization call from the application that will ask for the addresses of the Record buffers. These addresses are the starting addresses of the A and B buffers, which are concatenated back-to-back. See *Figure 2: Play and Record Buffers* for a conceptual drawing. The driver then returns the pointer to the pointer of the start addresses.

API Name: GetChannelVolumeBufferAddresses
Direction: Application to Driver
Com Type: Host
OTF?: No
Var I/O: out: *char* **v

The driver returns a pointer to the starting address of the Channel Volume Control Buffer.

API Name: GetChannelRoutingBufferAddresses
Direction: Application to Driver
Com Type: Host
OTF?: No
Var I/O: out: *char* **r

The driver returns a pointer to the starting address of the Channel Routing Control Buffer.

API Name: GetTimecodeAddresses
Direction: Application to Driver
Com Type: Host
OTF?: No
Var I/O: out: *char* **t

The driver returns a pointer to the starting address of the ADAT Timecode Buffer.

API Name: GetRecPlayLatency
Direction: Application to Driver
Com Type: Host
OTF?: No
Var I/O: None

The driver will return the value <16> which is the number of sample frames that the record buffer will lag behind the play buffer. This latency description is covered in section 4.6. This value will only change when a new driver is released, if at all.

6.3 Calls that directly affect buffer fills

API Name: SetFillRoutine
Direction: Application to Driver
Com Type: Host
OTF?: No
Var I/O: in: *FillRoutine, in: *long* param

During the initialization phase, the application establishes a method of communication between the driver and fill routine of the app. When the hardware realizes that the play buffers are half empty, it interrputs the driver. The driver then calls the application's fill routine and passes the *param* variable everytime the hardware needs more data. The hardware will ring the host doorbell to signify the beginning of this process. See Request_for_data below.

API Name: RequestForData
Direction: Hardware to Driver
Com Type: PCI Interrupt; Doorbell and Mailbox
OTF?: Yes

The DSP DMAs data from the host Playback Buffers (A and B). It also DMAs into the Record buffer. The DSP begins outputting samples starting at the very first Playback Buffer A location. It then keeps track of how many frames it has played back and will issue a PCI interrupt when it has played completely through the first buffer.

PROCEDURE:

- The hardware rings the Local-to-PCI doorbell which generates a PCI interrupt
- The driver reads the doorbell (1 byte) into a register, and re-writes it back out to the doorbell, thus clearing the interrupt.
- If the doorbell is a positive number (MSB not set) then there are no errors; everything is OK

- If the doorbell is a negative number (MSB set) then there has been an error. The driver issues a **Stop_Play** command thus stopping the hardware playback
- The value of the doorbell determines the error-type; the driver has halted the hardware, and has retained the value of the doorbell, and has set its **state** to *stop*
- IF there are no errors, then the driver will call the applications Fill_Routine and pass the *param* variable as needed.