SoundLink DRS
Digital Recording Systems

1212 I/O

PCI Multi-Channel Audio Interface

Developer API

KORG
Professional Audio

1.0 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. 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 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 is an example of one Audio Data Frame as it relates to one of the playback buffers (the top of the next buffer is shown also):

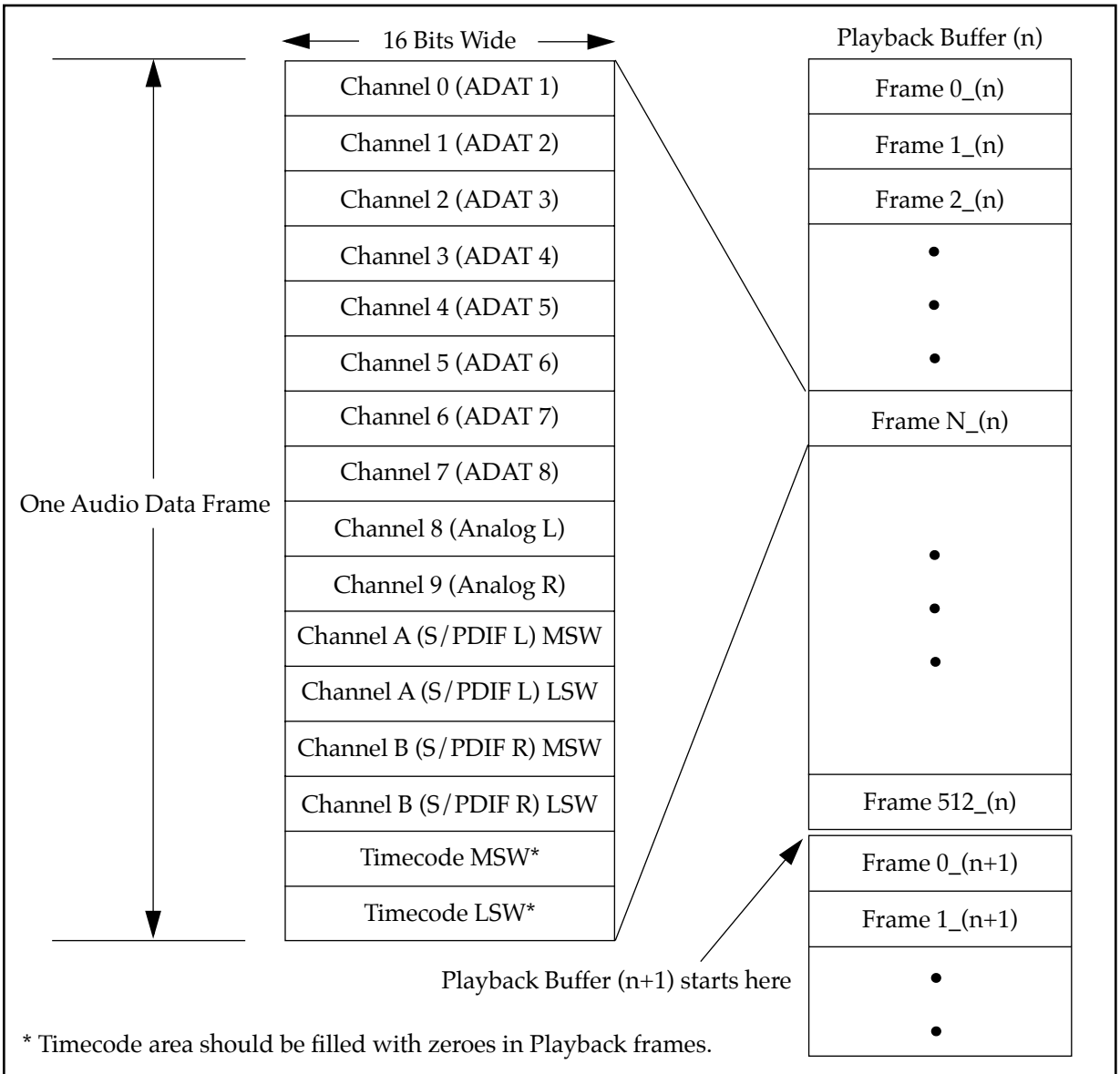

Figure 1: Play and Record Buffers

1.1 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):

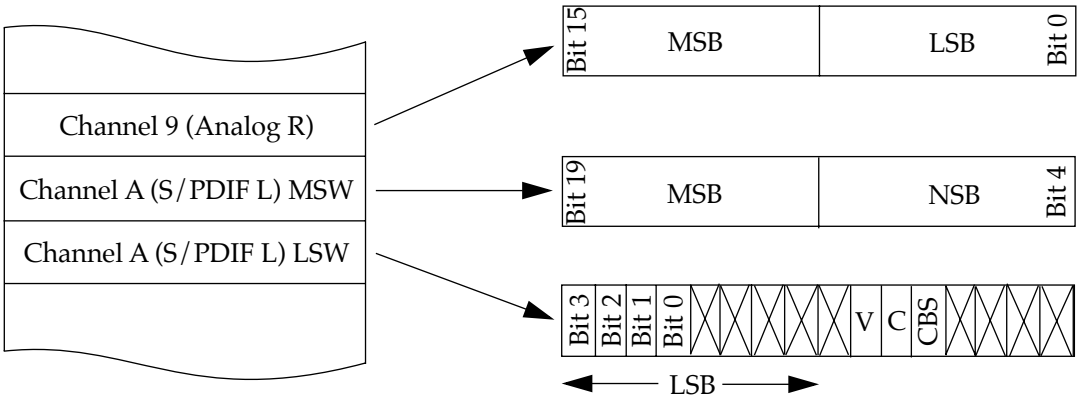

Figure 2: S/PDIF 20 bit data and V, U, C,P Locations

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 MSB word slot (Bits 19:4) and fill the LSB with zeros.

1.2 Record/Playback Latency Issues

1.2.1 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 set up the DMA starting addresses. After that, the DSP waits for a "trigger" signal (a call made through the driver); when the trigger is received, 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.

1.2.2 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 3 below:

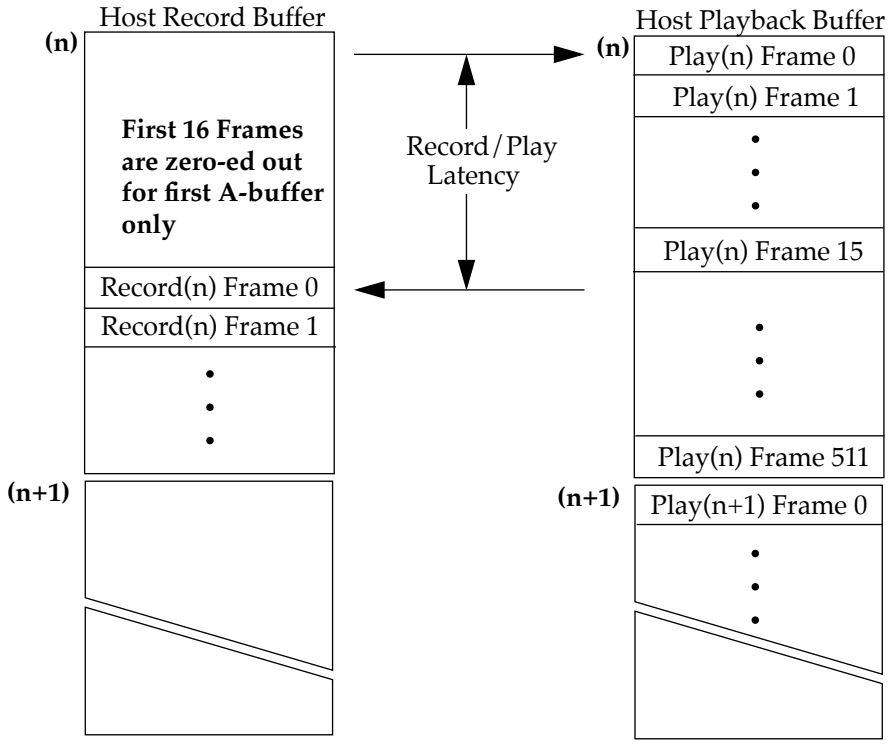

Figure 3: Record/Playback latency

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.

2.0 Adat Timecode Synchronization

2.1 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 allows applications to directly read ADAT timecode. This value is a 32 bit number, representing the number of sample clocks that have passed since the beginning of the tape. Every four audio data frames, the DMA sequence will result in a freshly updated timecode value. The application can translate the ADAT timecode into a more user-friendly SMPTE-style format (hours:minutes:seconds:frames). This can be calculated by dividing the ADAT timecode value (the number of elapsed sample clocks) by the current sample rate (48 or 44.1kHz), and subtracting the initial 2-minute "data" section (see section 2.1.3, below).

The timecode is read in two different ways, depending upon the current state of the 1212 I/O. During playback/record (when the 1212 I/O is in Play mode), the timecode comprises the last 4 bytes of the record buffer's audio data frames (as shown in figure 1, on page 2). When the 1212 I/O is in Stop mode, the current timecode may be read using the call, "GetADATTimeCode" (as described on page 10).

There are several important considerations when looking at ADAT timecode:

2.1.1 Timecode cannot be read when using internal clock

ADAT timecode can only be read when using the ADAT optical input, S/PDIF input, or Word Clock input as the master word clock; it cannot be read when using the internal clock (a limitation due to the AUSY2 ADAT interface chip).

2.1.2 Converting from ADAT timecode to hours:minutes:seconds

While the original ADAT and the ADAT XT display the same values at 48kHz, they show different values at 44.1kHz, as described below; the application developer should be aware of this, as it is a possible source of user confusion.

- The original ADAT's front-panel clock display is always based on a 48kHz sample rate. 44.1kHz operation is achieved by using the pitch controls to varispeed down by -147 cents (at which point the front panel switches from showing cents to displaying, "44.1"). Regardless of this setting, the clock display on the face of the ADAT remains based on a 48kHz rate, so that at 44.1kHz the seconds in the display pass more slowly than real-time seconds.
- On the ADAT XT and other XT-compatible machines, sample rate can be explicitly set to either 44.1kHz or 48kHz. The clock display on the face of the ADAT will adjust properly to the selected rate.

Note that the ADAT timecode is directly connected to the word clock, so that as the sample rate is changed (including varispeed settings), the ADAT timecode speeds up or slows down accordingly. This means that the ADAT timecode value (in elapsed samples) for a given physical location on the tape is always the same, regardless of sample rate or elapsed "real time."

2.1.3 ADAT timecode includes 2-minute "data" section

The ADAT tape format includes a "data" section at the head of each tape, before the main audio section begins. At 48kHz, this data section is 2 minutes long. The timecode values include these 2 minutes in the total number of elapsed samples, so that the start of the audio section of the tape–which is displayed as 00:00:00 on the front panel of the ADAT–has a timecode value of 0x0057 E400 (2 minutes at 48kHz). 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:frames value.

2.2 Fast ADAT Stop Mode

The 1212 I/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 1212 I/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 1212 I/O will also stop.

In order to hit the worst case of 65,535 clocks, the 1212 I/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 1212 I/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.

3.0 Application to Korg 1212 I/O API

3.1 Korg 1212 I/O Driver Communications

Upon system bootup, the Mac OS loads a copy of the Korg 1212 I/O driver for each Korg 1212 I/O card found in the system's PCI bus. Applications which want to communicate effectively with any Korg 1212 system should locate and open each of the installed drivers for 1212 I/O cards.

Note that although only one copy of the Korg 1212 I/O driver is needed in the Extensions folder, the Mac OS will load multiple copies of the driver: one for each card found in the PCI bus.

Each located driver will provide a reference number with which to identify a particular 1212 I/O card. The process by which a driver is opened returns an application client identification number, which the application should then use in all subsequent communications to the particular driver for a card.

Our sample code simplifies this process by defining a routine which fills up a data structure with all of the relevant information needed for each card in the system. Calling applications only need to determine whether any or all of the Korg 1212 I/O cards have been found and then implement the appropriate U/I and communication calls to address the 1212 I/O cards.

Applications should check whether the number of devices found equals the maximum number of device information elements allocated and if so, allocate a larger number and repeat the process until the number of devices returned is smaller than that of the number of devices allocated by the application. For an example of how to do this, refer to the Korg1212Signup() and OpenKorgDrivers() procedure calls.

The Korg 1212 I/O driver allows multiple active client applications for each card. This means that applications should check the OSErr value returned from calls to the driver in order to verify that normal processing of driver calls has taken place and handle things appropriately otherwise.

The sample code contains examples to implement every call in the Application-to-Korg 1212 I/O API.

Talking to a specific Korg 1212 I/O card involves deciding which card the application wants to communicate with and setting the appropriate ioCRefNum value in a parameter block structure. Then the client application reference number ioVRefNum and routine selector code csCode should be set. Any required parameters should be set in the csParam parameter array and a standard Mac OS device driver PBControlSync should be issued. An error code is returned by the driver and this value should be check to insure that the operation requested was performed correctly. This is demonstrated in the “Korg Driver commands” section of the sample code file.

3.2 1212 I/O Essentials

The 1212 I/O transfers data to and from the computer in packets. These packets are typically about 256 samples long per channel and get written to one of two buffers. The 1212 I/O generates a slot interrupt when it sees that one of its buffers is empty, and the application fills the buffer with new data. In this slot interrupt the application also updates the volume and pan settings for each channel. Volume and pan are usually integer values.

3.3 Application Audio Driver API

Rather than specifically having calls that set volume or send a packet of data, this API asks for addresses. This mechanism avoids having to call driver routines in real-time (which is very slow as it has to go through the trap dispatching mechanism in the Macintosh).

3.4 Definition of “session”

For this document, the phrase “session” means a record/playback event. For example, the user decides on the channel routing scheme before a session. The user hits play/record, etc., and creates some audio events. Later, he decides to re-map the I/O. This will start a new session. Once a session has been initiated,

the user can only adjust volumes going to the 1212 I/O–he cannot change routing without first stopping the session. This is the only on-the-fly change that can be made.

3.5 API Calls

This API lists the calls that audio software applications should make to the Korg 1212 I/O Multi-Channel Audio Interface Card.

3.5.1 Device Driver Management

OSErr Korg1212Signup(KorgCardsInfoPtr clientKorgInfoptr);

This call fills up the input KorgCardsInfo structure with the number of Korg 1212 I/O cards found. The number of cards found is set in the ionumCards field of the clientKorgInfo parameter. The cardsInfo field will contain a pointer to an array of Korg1212Info structures. This array will contain ionumCards elements. Subsequent API calls will require a pointer to one of these Korg1212Info structures as an input in order to address the correct Korg 1212 I/O device.

Korg1212InfoPtr device parameter

The Korg1212InfoPtr parameter must be passed to all of the remaining API function calls in order to determine which of the Korg 1212 I/O devices is being addressed by the call. The application must pass one of the legal Korg1212InfoPtr values returned through the clientKorgInfoptr structure pointer.

```
*****************************
KorgCardsInfo  gOurInfo;
short          i;

Korg1212Signup(&gOurInfo);    // signup and get info structures

for (i = 0; i < gOurInfo.ionumCards; i++)
   SetupForPlay(&gOurInfo.cardsInfo[i]);    // set all cards to play
*****************************
```

OSErr Korg1212Signout(KorgCardsInfoPtr clientKorgInfoptr);

Signout application from all cards. The parameter clientKorgInfoptr should be the same parameter passed to Korg1212Signup. The cardsInfo field is disposed of and nulled out. Calls to each card driver are made in order to sign the client application out of the driver, which causes card audio settings to be restored to the selected user defaults. Note that further calls with the given clientKorgInfoptr will no longer be recognized by the driver.

The call used internally by Korg1212Signout to sign the application out from each card is Korg1212SignClientOut. This call should only be used by applications which find it necessary to manage their own KorgCardsInfo data structure and memory.

3.5.2 Device Configuration

OSErr GetPacketSize(Korg1212InfoPtr device, short *psize);

Returns the packet size in number of samples. This value is currently 512 samples; however, it may be changed in the future.

OSErr GetSampleSize(Korg1212InfoPtr device, short *sample_size);

Returns the size of one sample in bytes. Thus a returned value of one would correspond to 8-bit samples, a returned value of 2 would correspond to 16-bit samples etc. The 1212 I/O supports

S/PDIF data at either 16- or 20-bit depths; Adat and analog i/o are 16-bit only.

OSErr SetClockSourceRate(Korg1212InfoPtr device, long srate);

Sets the system/device (hardware) sample rate and clock source. Use the following:

Table 1: Clock Source/Rate vs. srate Variable

Clock Source/Rate	srate
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

The clock source and rate can only be called at the beginning of a session. Changing the clock/source is done while the card is in "stop" state, and will result in a brief glitch of muted audio.

OSErr GetClockSourceRate (Korg1212InfoPtr device, long *srate);

Gets the system/device (hardware) clock source and sample rate. See Table 1, "Clock Source/Rate vs. srate Variable," above.

OSErr GetPlayChannelCount (Korg1212InfoPtr device, short *count);

Gets the number of mono playback channels in the system.

OSErr GetRecordChannelCount (Korg1212InfoPtr device, short *count);

Gets the number of mono input channels in the system.

OSErr SetInputGain (Korg1212InfoPtr device, short llevel, short rlevel);

This sets the attenuation for the analog input pad. 0 is no attenuation (full gain), 0x00FF is maximum attenuation (muted). This may be done on the fly, during a session. The left and right channel levels may be set independently; llevel is the left channel, rlevel is the right.

OSErr GetInputGain (Korg1212InfoPtr device, short *llevel, short *rlevel);

This call returns the analog pad attenuation values. 0 is no attenuation (full gain), 0x00FF is maximum attenuation (muted). The left and right channel levels may be set independently; llevel is the left channel, rrlevel is the right.

3.5.3 Playback, Recording and Monitoring Calls

OSErr GetPlayBufferAddresses (Korg1212InfoPtr device, char ***buffers);

Returns a pointer to an array of playback buffers, where the number of buffers can be determined by calling Korg1212GetBufferCount. The buffers consist of concatenated audio data frames, as defined in *Figure 1*: Play and Record Buffers.

OSErr GetRecordBufferAddresses (Korg1212InfoPtr device, char ***buffers);

Returns two pointers to the record buffers. Buffer format is the same as in previous call.

OSErr GetBufferCount(Korg1212InfoPtr device, short *count);

Get the number of playback/record buffers used by the 1212 card. Note that this number used to be hardwired to 2.

OSErr GetChannelVolumeBufferAddresses (Korg1212InfoPtr device, char **b1);

Returns a pointer to the channel volume buffer. This buffer is 12 words long, each word corresponds to a channel volume control. The application will write to this buffer each time it needs to change the volume settings.

OSErr GetChannelRoutingBufferAddresses (Korg1212InfoPtr device, char **b1);

Returns a pointer to the channel routing buffer. This buffer is 12 words long, each word corresponds to a channel routing control. The application will write to this buffer each time it needs to change the routing, which is only at the beginning of a session.

OSErr GetAdatTimecodeAddresses (Korg1212InfoPtr device, char **b1);

Returns a pointer to the timecode register. This register contains the Alesis 32 bit timecode value. It is written automatically each time the hardware does a DMA cycle.

OSErr GetADATTimeCode(Korg1212InfoPtr device, long *timecode);

Get the current ADAT timecode received and decoded by the 1212 I/O card. Use this call to determine the last known ADAT time location while in Stop mode. While in play mode the current ADAT timecode is kept in the ADAT miscellaneous memory location.

OSErr Korg1212SetFillRoutine(Korg1212InfoPtr device, OSErr (*FillRoutine) (long param), long param, long interrupt);

Sets the address of the packet-interrupt callback routine. The “param” parameter is expected to be passed as a parameter to the callback routine. The packet-interrupt routine is expected to be called every time the audio device desires more data.

This routine has been extended to allow clients to specify the interrupt level at which the callback is executed. Two independent callbacks are now supported. The interrupt parameter specifies the callback level:

```
enum {
kPRIMARYCALL= 0,
kSECONDARYCALL= 1,
kSECONDARYSWCALL= 2
};
```

kPRIMARYCALL - specifies a callback which is executed at the hardware interrupt level. Note that other hardware devices are not being serviced while a callback at this level executes and thus callback routines should minimize the amount of time spent in them.

kSECONDARYCALL and kSECONDARYSWCALL - specify a callback which is executed at one of two secondary
interrupt levels: secondary interrupt handlers or software task interrupts. Secondary interrupt handlers do not preclude primary interrupt processing routines and thus can spend more time executing client tasks. Software task interrupt callbacks are similar but they are scheduled from a system-run software interrupt and are compatible with Virtual Memory (whereas secondary interrupts
are not.)

Note that both primary level and secondary level direct and UPP callbacks can be specified for each client. This means that up to 4 callbacks can be independently set by each client. Secondary interrupts are set to either kSECONDARYCALL or kSECONDARYSWCALL but not both.

OSErr Set1212UPPFillRoutine(short theclient, UniversalProcPtr theroutine, long param, long interrupt);

Set a Universal Procedure Pointer (UPP) callback from the 1212 driver. This will allow compatibility with 68k code which needs to set up callback routines. The parameters are similar to those in the Set1212FillRoutine function call.

OSErr SetupForPlay (Korg1212InfoPtr device);

Called after the playback buffers have been pre-initialized with data. The 1212 I/O can call the packet-interrupt routine at this point. Any setup that needs to happen before playback occurs should happen during this call.

OSErr TriggerFromADAT (Korg1212InfoPtr device, long adattimecode, short adatstopmode);

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. "adattimecode" contains the Alesis 32 bit timecode value, and "adatstopmode" contains either a value of 0x8000 for fast stop mode or 0 for non-fast stop mode.

OSErr TriggerPlay (Korg1212InfoPtr device);

Start playback immediately. All initialization for playback should have happened in "SetupForPlay".

OSErr StopPlay (Korg1212InfoPtr device);

Stops playback or monitor-only mode.

OSErr MonitorOn (Korg1212InfoPtr device, short newmode);

Turns monitor mode on or off depending on the "Boolean" input parameter. Monitor mode is always on in playback mode depending on parameter settings.

OSErr LockBuffers (Korg1212InfoPtr device);

Locks Playback/Record buffers. Prior to calling SetupForPlay, applications should insure the exclusive use of the playback/record buffers by locking them by means of this call. This allows an application needing access to the playback/record buffers for any purpose (as in pre-filling playback buffers) to guarantee that other applications are excluded from accesing the buffers. Note that nothing physically prevents another client application from accessing the buffers, this works on a courtesy principle only. Applications should relinquish the buffers in order to allow other applications to playback and record as well. See the Korg1212UnlockBuffers call below.

An ideal protocol for locking and unlocking buffers would be to lock the buffers whenever the application is in the foreground and in playback/record ready mode and to unlock them whenever it goes into the background.

OSErr UnlockBuffers(short theclient);

Unlock Playback/Record buffers. Applications should relinquish previously locked buffers in order to allow other applications access to playback and record buffers. See the LockBuffers call above.

3.5.4 Playback and Record Parameter Settings

OSErr GetRecordPlayLatency (Korg1212InfoPtr device, short *latency);

Returns the value, in number of frames, of the record/playback latency that exists due to the buffering of data on the 1212 I/O card and the preloading that the DSP does to accommodate immediate playback upon trigger.

3.5.5 Other Settings

OSErr Calibrate (Korg1212InfoPtr device)

Initiates an ADC calibration cycle. The Korg driver takes care of this, including returning the cal value. Due to temperature effects on DC offsets, the user may wish to recalibrate the ADCs at any given time, not just on power up (in fact, the DC offset will probably change greatly after the first 10 minutes of system power. This can only be done before a session, not on-the-fly. It results in a brief glitch of muted audio.

4.0 Sample Code

```
/*
 * KorgDriverTest.c
 * Copyright © 1996 Korg R&D. All Rights Reserved.
 */

/*
    Korg 1212 I/O Driver Communications

    Upon system bootup, the Mac OS loads a copy of the Korg 1212 I/O driver
    for each Korg 1212 I/O card found in the system's PCI bus. Applications
    which want to communicate effectively with any Korg 1212 system should
    locate and open each of the installed drivers for 1212 I/O cards.

    Note that although only one copy of the Korg 1212 I/O driver is needed in
    the Extensions folder, the Mac OS will load multiple copies of the driver:
    one for each card found in the PCI bus.

    Each located driver will provide a reference number with which to identify
    a particular 1212 I/O card. The process by which a driver is opened returns
    an application client identification number, which the application should
    then use in all subsequent communications to the particular driver for a
    card.
    Our sample code simplifies this process by defining a routine which fills
    up a data structure with all of the relevant information needed for each
    card in the system. Calling applications only need to determine whether
    any or all of the Korg 1212 I/O cards have been found and then implement
    the appropriate U/I and communication calls to address the 1212 I/O cards.
    Applications should check whether the number of devices found equals
    the maximum number of device information elements allocated and if so, allocate
    a larger number and repeat the process until the number of devices returned is
    smaller than that of the number of devices allocated by the application.
    For an example of how to do this, refer to the Korg1212Signup() and OpenKorgDrivers()
    procedure calls.
    The Korg 1212 I/O driver allows multiple active client applications for each
    card. This means that applications should check the OSErr value returned
    from calls to the driver in order to verify that normal processing of driver
    calls has taken place and handle things appropriately otherwise.

    The sample code contains examples to implement every call in the
    Korg 1212 I/O API.

    Talking to a specific Korg 1212 I/O card involves deciding which card
    the application wants to communicate with and setting the appropriate
    ioCRefNum value in a parameter block structure. Then the client application
    reference number ioVRefNum and routine selector code csCode should be set.
    Any required parameters should be set in the csParam parameter array and
    a standard Mac OS device driver PBControlSync should be issued.
    An error code is returned by the driver and this value should be check
    to insure that the operation requested was performed correctly.
    This is demonstrated in the "Korg Driver commands" section of the sample
    code file.
*/


/*  .________________________________________________________________________.
    | This is a very simple test program for the Korg 1212 PCI driver. |
    .________________________________________________________________________.
```

```
 */

#include "Korg1212DriverPrivate.h"

#include <ToolUtils.h>
#include <SegLoad.h>

#ifndef FALSE
#define FALSE     0
#define TRUE      1
#endif

#define kDriverNamePString"\p.Korg1212IODriver"/* Driver name: Pascal string*/
#define kDriverNameCString".Korg1212IODriver"/* Driver name: C string*/
#define kCreatorType'KORG'   /* Registered Creator Type*/

#define MAXPCICARDS32


/* Structure definitions */


/* CntrlParam is defined by Mac types
   it is included here for expediency in reading the API interface

struct CntrlParam {
    QElemPtr             qLink;
    short                    qType;
    short                    ioTrap;
    Ptr                          ioCmdAddr;
    IOCompletionUPP   ioCompletion;
    OSErr                    ioResult;
    StringPtr            ioNamePtr;
    short                    ioVRefNum;
    short                    ioCRefNum;
    short                    csCode;
    short                    csParam[11];
};
typedef struct CntrlParam CntrlParam, *CntrlParamPtr;
*/

struct Korg1212Info {
    short         krefNum;
    short         kClientID;
    Str255        kDriverName;
    RegEntryID    kdeviceID;
    PCIBusNumber  cardbusnumber;
    PCIDeviceFunction carddevicenumber;
};
typedef struct Korg1212Info Korg1212Info, *Korg1212InfoPtr;

struct KorgCardsInfo {
    short                    ionumCards;
    Korg1212InfoPtr   cardsInfo;
};
typedef struct KorgCardsInfo KorgCardsInfo, *KorgCardsInfoPtr;

/*
 * Defined PBControl csCodes. Note that we don't support all of these.
```

```
 */
/*
enum {
    kControlGoodbye= (-1),// Last call before close (ignored)
    kControlKillIO= 1,// This is handled by the Driver Manager
    kSetClockSourceRate= 2,
    kGetClockSourceRate= 3,
    kGetPacketSize= 4,
    kGetSampleSize= 5,
    kGetPlayCount= 6,
    kGetRecordCount= 7,
    kConfigureBufferMemory= 8,
    kConfigureMiscMemory= 9,
    kGetPlayBuffAddress= 10,
    kGetRecBuffAddress= 11,
    kGetChanVolBuffAddress = 12,
    kGetChanRoutBuffAddress= 13,
    kGetTimecodeAddress= 14,
    kGetRecPlayLatency= 15,
    kSetFillRoutine= 16,
    kSetCalibrationValue= 17,
    kCalibrate    = 18,
    kSetMonitorStatus= 19,
    kSetupForPlay= 20,
    kTriggerFromADAT= 21,
    kTriggerPlay= 22,
    kStopPlay     = 23,
    kSetInputGain= 24,
    kGetInputGain= 25,
    kRequestForData= 26,
    kLockBuffers= 29,
    kUnlockBuffers= 30,
    kGetADATTimeCode= 32
};
*/

/* Error codes returned by the application */

#define gMemAllocationError(-100)
#define gDriverOpenError(-101)
#define gNoDriverError(-102)

/* Error codes returned by the driver */

#define kDriverMemAllocErr(-1)
#define kDriverTooManyClientsErr(-2)
#define kDriverNoDriverFile(-3)
#define kDriverNoDSPCodeResource(-4)
#define kDriverNoDSPCodeMemory(-5)
#define kDriverConfigMemError(-6)
#define kDriverConfigMiscError(-7)
#define kDriverNoSuchClientError(-8)
#define kAlreadyInPlay(-9)
#define kBadRateValue(-10)
#define kAlreadyLocked(-11)

/* global variables used in our sample */

short                gDriverRefNum;
```

```
short                gClientNum;
KorgCardsInfo gKorg1212Info;

CntrlParam        gCtrlParam;

/* Function prototypes */

/*
OSErr  Korg1212Signup(void);
void   OpenKorgDrivers(KorgCardsInfoPtr theinfo, long maxdevices);
OSErr  OpenNextDevice(KorgCardsInfoPtr theinfo, long maxdevice, short *curunitindex);
*/

/* Sample Signup
*/
OSErr OurApplicationSignup(void)
{
    OSErrstatus;

    status = Korg1212Signup(&gKorg1212Info);
    if (status != noErr) {
    // process error
    return(status);
    }

    if (gKorg1212Info.ionumCards) {
        // Our sample app only talks to the first card found..
        // Other applications may want to put up a popup menu to select
        // cards in operation by using the carddevicenumber field
        gDriverRefNum = gKorg1212Info.cardsInfo[0].krefNum;
        gClientNum = gKorg1212Info.cardsInfo[0].kClientID;
    }
    return(noErr);
}

/*
 * Korg Device Driver management...
 */
OSErr Korg1212Signup(KorgCardsInfoPtr clientKorgInfoptr)
{
    long maxPCIdevices;

    maxPCIdevices = MAXPCICARDS;// Allocate device info structs
    clientKorgInfoptr->cardsInfo =
        (Korg1212InfoPtr) NewPtr(maxPCIdevices*sizeof(Korg1212Info));
    if (clientKorgInfoptr->cardsInfo == NULL) return(gMemAllocationError);
                                    // Now get device drivers and open them
    OpenKorgDrivers(clientKorgInfoptr, maxPCIdevices);
    while (clientKorgInfoptr->ionumCards == maxPCIdevices) {
        maxPCIdevices += MAXPCICARDS;// too many devices in the system
        DisposePtr((Ptr) clientKorgInfoptr->cardsInfo);// deallocate previous info
        clientKorgInfoptr->cardsInfo =
            (Korg1212InfoPtr) NewPtr(maxPCIdevices*sizeof(Korg1212Info));
        if (clientKorgInfoptr->cardsInfo == NULL) return(gMemAllocationError);
                                    // try the allocation again...
        OpenKorgDrivers(clientKorgInfoptr, maxPCIdevices);
    }
    return(noErr);
}
```

```
/*
 * Find all Korg device drivers loaded by the Mac OS...
 */
static void
OpenKorgDrivers(KorgCardsInfoPtr theinfo, long maxdevices)
{
    OSErr         status;
    short         i;
    short         theUnitIndex;

    // Initialize Driver info
    theinfo->ionumCards = 0;
    for (i = 0; i < maxdevices; i++) theinfo->cardsInfo[i].krefNum = 0;

    for (theUnitIndex = 0; (status = OpenNextDevice(theinfo, maxdevices, &theUnitIndex)) ==
noErr; theUnitIndex++) {
        if (status != noErr) {
        }
        else {
        }
    }
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * This is a generic function that retrieves a property from the Name Registry,
 * allocating memory for it with NewPtr. It looks in the Name Registry entry
 * for this driver -- the DriverInitializeCmd passed this as one of its parameters.
 */
static OSErr
GetThisProperty(
        RegEntryIDPtrregEntryIDPtr,/* Driver's Name Registery ID*/
        RegPropertyNamePtrregPropertyName,
        RegPropertyValue**regPropertyValuePtr,
        RegPropertyValueSize*regPropertyValueSizePtr
    )
{
    OSErr             status;

    /*
     * In addition to getting the size of a property, this function will fail if
     * the property is not present in the registry. We NULL the result before
     * starting so we can dispose of the property even if this function failed.
     */
    *regPropertyValuePtr = NULL;
    status = RegistryPropertyGetSize(
                regEntryIDPtr,
                regPropertyName,
                regPropertyValueSizePtr
            );
    //** CheckStatus(status, "\pRegistryPropertyGetSize failed");
    if (status == noErr) {
        *regPropertyValuePtr = (RegPropertyValue *)
                    NewPtr(*regPropertyValueSizePtr);
        status = RegistryPropertyGet(
                    regEntryIDPtr,
                    regPropertyName,
                    *regPropertyValuePtr,
```

```
                regPropertyValueSizePtr
            );
    }
    return (status);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Dispose of a property that was obtained by calling GetThisProperty(). This sample is
 * specific to applications: the property was allocated with NewPtr.
 */
static void
DisposeThisProperty(
        RegPropertyValue*regPropertyValuePtr
    )
{
    if (*regPropertyValuePtr != NULL) {
        DisposePtr(*regPropertyValuePtr);
        *regPropertyValuePtr = NULL;
    }
}

/*
 * Find the next device (after this one) that has our device name.
 */
static OSErr
OpenNextDevice(KorgCardsInfoPtr theinfo, long maxdevice, short *curunitindex)
{
    OSErr             status;
    short             index;
    ItemCount     items;
    DriverRefNumrefNum;
    UnitNumber    unusedUnitNumber;
    DriverFlags   driverFlags;
    DriverOpenCountdriverOpenCount;
    RegEntryID    deviceID;
    CFragHFSLocatorcFragHFSLocator;
    CFragConnectionIDcFragConnectionID;
    DriverEntryPointPtrfragmentMain;
    DriverDescriptiondriverDescription;
    RegPropertyValueSizeassignedAddressSize;
    RegPropertyValue*assignedAddressProperty; /* Assigned Address property*/
    PCIAssignedAddressPtrapciAssignedAddressPtr;/* Assigned Address element ptr*/
    Str255            gDriverName;

    for (;; *curunitindex += 1) {
        if (*curunitindex > HighestUnitNumber()) {
            status = badUnitErr;
            break;
        }
        items = 1;
        status = LookupDrivers(*curunitindex, *curunitindex, FALSE, &items, &refNum);
        if (status != noErr)
            break;
        if (items == 1) {
            /*
             * We have found a driver (but not necessarily ours).
             */
            status = GetDriverInformation(
                      refNum,
```

```
                        &unusedUnitNumber,
                        &driverFlags,
                        &driverOpenCount,
                        gDriverName,
                        &deviceID,
                        &cFragHFSLocator,
                        &cFragConnectionID,
                        &fragmentMain,
                        &driverDescription
                    );
            if (status != noErr)
                break;
            if (PStrCmp(gDriverName, kDriverNamePString) == 0) {
                /*
                 * This is our device. Try to open it. (This presumes that the
                 * driver was installed "load on discovery.")
                 */
                status = OpenInstalledDriver(refNum, 3); /* Allow read/write*/
                if (status < 0) {
                    return(gDriverOpenError);
                }
                else {
                    // Copy relevant information to device description struct
                    index = theinfo->ionumCards;
                    if (index < maxdevice) {
                        theinfo->cardsInfo[index].krefNum = refNum;
                        theinfo->cardsInfo[index].kClientID = status; // Status > 0 is client ID
number!
                        theinfo->cardsInfo[index].kdeviceID = deviceID;
                        BlockMove(gDriverName, theinfo->cardsInfo[index].kDriverName,
sizeof(Str255));
                        // Get PCI bus number for this card
                        assignedAddressProperty = NULL;
                        status = GetThisProperty(
                                    &deviceID,
                                    kPCIAssignedAddressProperty,
                                    &assignedAddressProperty,
                                    &assignedAddressSize
                                );
                        apciAssignedAddressPtr = (PCIAssignedAddressPtr)
assignedAddressProperty;
                        theinfo->cardsInfo[index].cardbusnumber = apciAssignedAddressPtr-
>busNumber;
                        theinfo->cardsInfo[index].carddevicenumber = (apciAssignedAddressPtr-
>deviceFunctionNumber)>>3;

                        DisposeThisProperty(assignedAddressProperty);
                        theinfo->ionumCards += 1;// increment drivers found
                    }
                    break;
                }
            } /* If this is our device */
        } /* If LookupDrivers found a device */
    } /* Unit number loop */
    return (status);
}

/*
 * Korg Driver commands...
```

```
 */

/* PBControl calls will pass inputs in csParam */
/* the driver reference number identifying each card*/
/* is passed in ioCRefNum and the client reference*/
/* number is passed in ioVRefNum*/
/* control codes specific for each call in csCode */
/* returned values will be in (long *) &PB.csParam[0]*/

/* Make a PBControl call to the 1212 I/O Driver */

static OSErr Make1212Call(CntrlParam *pbptr, short clientNum, short driverNum, short callcode)
{
    OSErr             status;
    unsigned chardummyBuffer;

    status = noErr;
    /*
     * Setup the parameter block
     */
    pbptr->ioCRefNum = driverNum;
    pbptr->ioVRefNum = clientNum;
    pbptr->csCode = callcode;
    pbptr->ioCmdAddr = (Ptr) &dummyBuffer;
    status = PBControlSync((ParmBlkPtr) pbptr);
    return ((status != noErr) ? status : pbptr->ioResult);
}


/*
 * Korg Driver API calls...
 */


/* Device Configuration */

/* Prototypes... */
OSErr Korg1212GetPacketSize(Korg1212InfoPtr device, short *psize);
OSErr Korg1212GetSampleSize(Korg1212InfoPtr device, short *sample_size);
OSErr Korg1212SetClockSourceRate(Korg1212InfoPtr device, long srate);
OSErr Korg1212GetClockSourceRate(Korg1212InfoPtr device, long *srate);
OSErr Korg1212GetPlayChannelCount(Korg1212InfoPtr device, short *count);
OSErr Korg1212GetRecordChannelCount(Korg1212InfoPtr device, short *count);
OSErr Korg1212SetInputGain(Korg1212InfoPtr device, short llevel, short rlevel);
OSErr Korg1212GetInputGain(Korg1212InfoPtr device, short *llevel, short *rlevel);

/* Get 1212 sample packet size. csCode = kGetPacketSize */

/* Get 1212 sample packet size. csCode = kGetPacketSize */

OSErr Korg1212GetPacketSize(Korg1212InfoPtr device, short *psize)
{
    OSErr             status;
    short             *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (short *) &PB.csParam[0];
```

```
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetPacketSize);
    *psize = (short) *dptr;
    return (status);
#undef PB
}

/* Get 1212 sample size. csCode = kGetSampleSize */

OSErr Korg1212GetSampleSize(Korg1212InfoPtr device, short *sample_size)
{
    OSErr             status;
    short             *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (short *) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetSampleSize);
    *sample_size = (short) *dptr;
    return (status);
#undef PB
}

/* Set 1212 I/O source clock rate. csCode = kSetClockSourceRate */

OSErr Korg1212SetClockSourceRate(Korg1212InfoPtr device, long srate)
{
    OSErr             status;
#define PB(gCtrlParam)

    CLEAR(PB);
    PB.csParam[0] = (short) srate;
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kSetClockSourceRate);
    return (status);
#undef PB
}

/* Get 1212 I/O source clock rate. csCode = kGetClockSourceRate */

OSErr Korg1212GetClockSourceRate(Korg1212InfoPtr device, long *srate)
{
    OSErr             status;
    short             *dptr, temp;
#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (short *) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetClockSourceRate);
    temp = (short) *dptr;
    *srate = temp;
    return (status);
#undef PB
}

/* Get 1212 mono playback channel count. csCode = kGetPlayCount */

OSErr Korg1212GetPlayChannelCount(Korg1212InfoPtr device, short *count)
{
    OSErr             status;
```

```
    short             *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (short *) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetPlayCount);
    *count = (short) *dptr;
    return (status);
#undef PB
}

/* Get 1212 mono record channel count. csCode = kGetRecordCount */

OSErr Korg1212GetRecordChannelCount(Korg1212InfoPtr device, short *count)
{
    OSErr             status;
    short             *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (short *) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetRecordCount);
    *count = (short) *dptr;
    return (status);
#undef PB
}

/* Set 1212 input gain levels. csCode = kSetInputGain */

OSErr Korg1212SetInputGain(Korg1212InfoPtr device, short llevel, short rlevel)
{
    OSErr             status;
    short             *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (short *) &PB.csParam[0];
    *dptr++ = llevel;
    *dptr++ = rlevel;
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kSetInputGain);
    return (status);
#undef PB
}

/* Get 1212 input gain levels. csCode = kGetInputGain */

OSErr Korg1212GetInputGain(Korg1212InfoPtr device, short *llevel, short *rlevel)
{
    OSErr             status;
    short             *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (short *) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetInputGain);
```

```
    *llevel = (short) *dptr++;
    *rlevel = (short) *dptr;
    return (status);
#undef PB
}

/* Playback, Recording and Monitoring... */

/* Prototypes... */
OSErr Korg1212GetPlayBufferAddresses(Korg1212InfoPtr device, char **b1, char **b2);
OSErr Korg1212GetRecordBufferAddresses(Korg1212InfoPtr device, char **b1, char **b2);
OSErr Korg1212GetChannelVolumeBufferAddresses(Korg1212InfoPtr device, char **b1);
OSErr Korg1212GetChannelRoutingBufferAddresses(Korg1212InfoPtr device, char **b1);
OSErr Korg1212GetAdatTimecodeAddresses(Korg1212InfoPtr device, char **b1);
OSErr Korg1212SetFillRoutine(Korg1212InfoPtr device, OSErr (*FillRoutine) (long param), long
param);
OSErr Korg1212SetupForPlay(Korg1212InfoPtr device);
OSErr Korg1212LockBuffers(Korg1212InfoPtr device);
OSErr Korg1212UnlockBuffers(Korg1212InfoPtr device);
OSErr Korg1212GetADATTimeCode(Korg1212InfoPtr device, long *timecode);
OSErr Korg1212TriggerPlay(Korg1212InfoPtr device);
OSErr Korg1212TriggerFromADAT(Korg1212InfoPtr device, long adattimecode, short adatstopmode);
OSErr Korg1212StopPlay   (Korg1212InfoPtr device);
OSErr Korg1212MonitorOn  (Korg1212InfoPtr device, short newmode);

/* Addition to 3rd party API...

    Device Driver management section

OSErr Korg1212Signup(KorgCardsInfoPtr clientKorgInfoptr);

This call fills up the input KorgCardsInfo structure with the number of
Korg 1212 I/O cards found. The number of cards found is set in the
ionumCards field of the clientKorgInfoptr paramenter. The cardsInfo field
will contain a pointer to an array of Korg1212Info structures. This array
will contain ionumCards elements. Subsequent API calls will require a
pointer to one of these Korg1212Info structures as an input in order
to address the correct Korg 1212 I/O device.


*****************************
For example:
KorgCardsInfogOurInfo;
short      i;

Korg1212Signup(&gOurInfo); signup and get info structures

for (i = 0; i < gOurInfo.ionumCards; i++)
    Korg1212SetupForPlay(&gOurInfo.cardsInfo[i]);set all cards to play
*****************************


   Playback, Recording and Monitoring Calls section

OSErr Korg1212TriggerFromADAT(Korg1212InfoPtr device, long adattimecode, short adatstopmode);

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.
```

```
adattimecode contains the Alesis 32 bit timecode value and adatstopmode contains either
a value of 0x8000 for fast stop mode or 0 for non-fast stop mode.

*/


/* Get 1212 I/O Playback Buffer addresses. csCode = kGetPlayBuffAddress */

OSErr Korg1212GetPlayBufferAddresses(Korg1212InfoPtr device, char **b1, char **b2)
{
    OSErr             status;
    char              **dptr;

#define PB(gCtrlParam)

/* Get playback memory buffers... */
    status = noErr;
    dptr = (char **) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetPlayBuffAddress);
    *b1 = (char *) *dptr++;
    *b2 = (char *) *dptr;
    return (status);
#undef PB
}

/* Get 1212 I/O Record Buffer addresses. csCode = kGetRecBuffAddress */

OSErr Korg1212GetRecordBufferAddresses(Korg1212InfoPtr device, char **b1, char **b2)
{
    OSErr             status;
    char              **dptr;

#define PB(gCtrlParam)

    dptr = (char **) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetRecBuffAddress);
    *b1 = (char *) *dptr++;
    *b2 = (char *) *dptr;
    return (status);
#undef PB
}

/* Get 1212 I/O Channel Volume Buffer addresses. csCode = kGetChanVolBuffAddress */

OSErr Korg1212GetChannelVolumeBufferAddresses(Korg1212InfoPtr device, char **b1)
{
    OSErr             status;
    char              **dptr;

#define PB(gCtrlParam)

    dptr = (char **) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetChanVolBuffAddress);
    *b1 = (char *) *dptr;
    return (status);
#undef PB
}

/* Get 1212 I/O Channel Routing Buffer addresses. csCode = kGetChanRoutBuffAddress */
```

```
OSErr Korg1212GetChannelRoutingBufferAddresses(Korg1212InfoPtr device, char **b1)
{
    OSErr             status;
    char              **dptr;

#define PB(gCtrlParam)

    dptr = (char **) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetChanRoutBuffAddress);
    *b1 = (char *) *dptr;
    return (status);
#undef PB
}

/* Get 1212 I/O Channel Routing Buffer addresses. csCode = kGetTimecodeAddress */

OSErr Korg1212GetAdatTimecodeAddresses(Korg1212InfoPtr device, char **b1)
{
    OSErr             status;
    char              **dptr;

#define PB(gCtrlParam)

    dptr = (char **) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetTimecodeAddress);
    *b1 = (char *) *dptr;
    return (status);
#undef PB
}

/* Set 1212 playback fill callback routine. csCode = kSetFillRoutine */

OSErr Korg1212SetFillRoutine(Korg1212InfoPtr device, OSErr (*FillRoutine) (long param), long
param)
{
    OSErr             status;
    long              *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (long *) &PB.csParam[0];
    *dptr++ = (long) FillRoutine;
    *dptr = (long) param;
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kSetFillRoutine);
    return (status);
#undef PB
}

/* Set 1212 I/O Play Mode. csCode = kSetupForPlay */

OSErr Korg1212SetupForPlay(Korg1212InfoPtr device)
{
    OSErr             status;
#define PB(gCtrlParam)

    status = noErr;
```

```
    CLEAR(PB);
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kSetupForPlay);
    return (status);
#undef PB
}

/* Lock 1212 I/O play buffers for this client. csCode = kLockBuffers */

OSErr Korg1212LockBuffers(Korg1212InfoPtr device)
{
    OSErr             status;

#define PB(gCtrlParam)

    CLEAR(PB);
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kLockBuffers);
    return (status);
#undef PB
}

/* Unlock 1212 I/O play buffers for this client. csCode = kUnlockBuffers */

OSErr Korg1212UnlockBuffers(Korg1212InfoPtr device)
{
    OSErr             status;

#define PB(gCtrlParam)

    CLEAR(PB);
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kUnlockBuffers);
    return (status);
#undef PB
}

/* Get current ADAT timecode value for this client. csCode = kGetADATTimeCode */

OSErr Korg1212GetADATTimeCode(Korg1212InfoPtr device, long *timecode)
{
    OSErr             status;
    long              *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (long *) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetADATTimeCode);
    *timecode = (long) *dptr;
    return (status);
#undef PB
}

/* Set 1212 I/O to Play. csCode = kTriggerPlay */

OSErr Korg1212TriggerPlay(Korg1212InfoPtr device)
{
    OSErr             status;
#define PB(gCtrlParam)

    CLEAR(PB);
```

```
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kTriggerPlay);
    return (status);
#undef PB
}

/* Set 1212 I/O to trigger from ADAT. csCode = kTriggerFromADAT */

OSErr Korg1212TriggerFromADAT(Korg1212InfoPtr device, long adattimecode, short adatstopmode)
{
    OSErr             status;
    long              *dptr;
    short             *sptr;
#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (long *) &PB.csParam[0];
    *dptr++ = (long) adattimecode;// copy parameters...
    sptr = (short *) dptr;
    *sptr = (long) adatstopmode;
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kTriggerFromADAT);
    return (status);
#undef PB
}

/* Set 1212 I/O to Stop. csCode = kStopPlay */

OSErr Korg1212StopPlay(Korg1212InfoPtr device)
{
    OSErr             status;
#define PB(gCtrlParam)

    CLEAR(PB);
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kStopPlay);
    return (status);
#undef PB
}

/* Set 1212 I/O Monitor Status. csCode = kSetMonitorStatus */

OSErr Korg1212MonitorOn(Korg1212InfoPtr device, short newmode)
{
    OSErr             status;
#define PB(gCtrlParam)

    CLEAR(PB);
    PB.csParam[0] = newmode;
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kSetMonitorStatus);
    return (status);
#undef PB
}


/* Playback and Record Parameter Settings... */

/* Prototypes... */
OSErr Korg1212GetRecordPlayLatency(Korg1212InfoPtr device, short *latency);

/* Get 1212 play frame latency. csCode = kGetRecPlayLatency */
```

```
OSErr Korg1212GetRecordPlayLatency(Korg1212InfoPtr device, short *latency)
{
    OSErr             status;
    short             *dptr;

#define PB(gCtrlParam)

    CLEAR(PB);
    dptr = (short *) &PB.csParam[0];
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kGetRecPlayLatency);
    *latency = (short) *dptr;
    return (status);
#undef PB
}

/* Other Settings... */

/* Prototypes... */
OSErr Korg1212Calibrate  (Korg1212InfoPtr device);

/* Perform 1212 calibration. csCode = kCalibrate */

OSErr Korg1212Calibrate(Korg1212InfoPtr device)
{
    OSErr             status;

#define PB(gCtrlParam)

    CLEAR(PB);
    status = Make1212Call(&PB, device->kClientID, device->krefNum, kCalibrate);
    return (status);
#undef PB
}


/*
 *
 */
/* Sample code to set buffers and buffer values... */
/*
 *
 */

short*playAbuff;      /* Playback buffers*/
short*playBbuff;
short*recAbuff;       /* Record buffers*/
short*recBbuff;
short*chanVoladdress;/* Channel Volume buffer*/
short*chanRouteaddress;/* Channel Routing buffer*/
long*timebuff;        /* ADAT timecode buffer*/

/* Routines to set volume and routing values */

static void SetChannelVolume(short channel, short newvalue);
static void SetChannelVolume(short channel, short newvalue)
{
    chanVoladdress[channel] = EndianSwap16Bit(newvalue);
}
```

```
static void SetChannelRoute(short channel, short newvalue);
static void SetChannelRoute(short channel, short newvalue)
{
    chanRouteaddress[channel] = EndianSwap16Bit(newvalue);
}

/* Sample to show how to get buffer addresses */

static void GetAllBufferAddresses(void);
static void GetAllBufferAddresses(void)
{
    OSErr             status;
    Korg1212InfoPtrcurDevice;

    curDevice = &gKorg1212Info.cardsInfo[0];
/* Get all of the memory buffers... */
    status = Korg1212GetPlayBufferAddresses(curDevice, (char **) (&playAbuff), (char **)
(&playBbuff));
    status = Korg1212GetRecordBufferAddresses(curDevice, (char **) (&recAbuff), (char **)
(&recBbuff));
    status = Korg1212GetChannelVolumeBufferAddresses(curDevice, (char **) (&chanVoladdress));
    status = Korg1212GetChannelRoutingBufferAddresses(curDevice, (char **)
(&chanRouteaddress));
    status = Korg1212GetAdatTimecodeAddresses(curDevice, (char **) (&timebuff));
}
```