

Devices_Manual

COLLABORATORS

	<i>TITLE :</i> Devices_Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Devices_Manual	1
1.1	Amiga® RKM Devices: 2 Audio Device	1
1.2	2 Audio Device / About Amiga Audio	1
1.3	2 / About Amiga Audio / Definitions	2
1.4	2 Audio Device / Audio Device Commands and Functions	3
1.5	2 Audio Device / Device Interface	4
1.6	2 / Device Interface / Opening The Audio Device	5
1.7	2 / Device Interface / Audio Device Command Types	6
1.8	2 / Device Interface / Scope Of Audio Commands	6
1.9	2 / Device Interface / Audio And System I/O Functions	6
1.10	2 / Audio And System I/O Functions / BeginIO()	7
1.11	2 / Audio And System I/O Functions / Wait() and WaitPort()	7
1.12	2 / Audio And System I/O Functions / AbortIO()	7
1.13	2 / Device Interface / Closing The Audio Device	7
1.14	2 Audio Device / A Simple Audio Example	8
1.15	2 Audio Device / Audio Allocation and Arbitration	8
1.16	2 Audio Device / Allocation and Arbitration Commands	9
1.17	2 / Allocation and Arbitration Commands / ADCMD_ALLOCATE	10
1.18	2 / ADCMD_ALLOCATE / How ADCMD_ALLOCATE Operates	10
1.19	2 / ADCMD_ALLOCATE / The ADIOF_NOWAIT Flag	10
1.20	2 / ADCMD_ALLOCATE / ADCMD_ALLOCATE Examples	11
1.21	2 / ADCMD_ALLOCATE / The Allocation Key	11
1.22	2 / Allocation and Arbitration Commands / ADCMD_FREE	11
1.23	2 / Allocation and Arbitration Commands / ADCMD_SETPREC	12
1.24	2 / Allocation and Arbitration Commands / ADCMD_LOCK	12
1.25	2 Audio Device / Hardware Control Commands	13
1.26	2 / Hardware Control Commands / CMD_WRITE	13
1.27	2 / Hardware Control Commands / ADCMD_FINISH	14
1.28	2 / Hardware Control Commands / ADCMD_PERVOL	14
1.29	2 / Hardware Control Commands / CMD_FLUSH	14

1.30	2 / Hardware Control Commands / CMD_RESET	15
1.31	2 / Hardware Control Commands / ADCMD_WAITCYCLE	15
1.32	2 / Hardware Control Commands / CMD_STOP	15
1.33	2 / Hardware Control Commands / CMD_START	15
1.34	2 / Hardware Control Commands / CMD_READ	15
1.35	2 Audio Device / Double Buffered Sound Example	16
1.36	2 Audio Device / Additional Information on the Audio Device	16

Chapter 1

Devices_Manual

1.1 Amiga® RKM Devices: 2 Audio Device

The Amiga has four hardware audio channels - two of the channels produce audio output from the left audio connector, and two from the right. These channels can be used in many ways. You can combine a right and a left channel for stereo sound, use a single channel, or play a different sound through each of the channels to create four-part harmony.

- About Amiga Audio
- Audio Device Commands and Functions
- Device Interface
- A Simple Audio Example
- Audio Allocation and Arbitration
- Allocation and Arbitration Commands
- Hardware Control Commands
- Double Buffered Sound Example
- Additional Information on the Audio Device

1.2 2 Audio Device / About Amiga Audio

Most personal computers that produce sound have hardware designed for one specific synthesis technique. The Amiga computer uses a very general method of digital sound synthesis that is quite similar to the method used in digital hi-fi components and state-of-the-art keyboard and drum synthesizers.

For programs that can afford the memory, playing sampled sounds gives you a simple and very CPU-efficient method of sound synthesis. A sampled sound is a table of numbers which represents a sound digitally. When the sound is played back by the Amiga, the table is fed by a DMA channel into one of the four digital-to-analog converters in the custom chips. The digital-to-analog converter converts the samples into voltages that can be played through amplifiers and loudspeakers, reproducing the sound.

On the Amiga you can create sound data in many other ways. For instance, you can use trigonometric functions in your programs to create the more traditional sounds - sine waves, square waves, or triangle waves - by

using tables that describe their shapes. Then you can combine these waves for richer sound effects by adding the tables together. Once the data are entered, you can modify them with techniques described below. For information about the limitations of the audio hardware and suggestions for improving system efficiency and sound quality, refer to the Amiga Hardware Reference Manual.

Some commands enable your program to co-reside with other programs using the audio device at the same time. Programs can co-reside because the audio device handles allocation of audio channels and arbitrates among programs competing for the same resources. When properly used, this allows many programs to use the audio device simultaneously.

The audio device commands help isolate the programmer from the idiosyncrasies of the custom chip hardware and make it easier to use. But you can also produce sound on the Amiga by directly accessing the hardware registers if you temporarily lock out other users first. For certain types of sound synthesis, this is more CPU-efficient.

Definitions

1.3 2 / About Amiga Audio / Definitions

Terms used in the following discussions may be unfamiliar. Some of the more important ones are defined below.

Amplitude

The height of a waveform, which corresponds to the amount of voltage or current in the electronic circuit.

Amplitude modulation

A means of producing special audio effects by using one channel to alter the amplitude of another.

Channel

One "unit" of the audio device.

Cycle

One repetition of a waveform.

Frequency

The number of times per second a cycle repeats.

Frequency modulation

A means of producing special audio effects by using one channel to affect the period of the waveform produced by another channel.

Period

The time elapsed between the output of successive sound samples, in units of system clock ticks.

Precedence

Priority of the user of a sound channel.

Sample

Unit of audio data, one of the fixed-interval points on the waveform.

Waveform

Graph that shows a model of how the amplitude of a sound varies over time-usually over one cycle.

1.4 2 Audio Device / Audio Device Commands and Functions

Command -----	Operation -----
ADCMD_ALLOCATE	Allocate one or more of the four audio channels.
ADCMD_FINISH	Abort the current write request on one or more of the channels. Can be done immediately or at the end of the current cycle.
ADCMD_FREE	Free one or more audio channels.
ADCMD_LOCK	Lock one or more audio channels.
ADCMD_PERVOL	Change the period and volume for writes in progress. Can be done immediately or at the end of the cycle.
ADCMD_SETPREC	Set the allocation precedence of one or more channels.
ADCMD_WAITCYCLE	Wait for the current write cycle to complete on a single channel. Returns at the end of the cycle or immediately if no cycle is active on the channel.
CMD_FLUSH	Purge all write cycles and waitcycles (in-progress and queued) for one or more channels.
CMD_READ	Return a pointer to the I/O block currently writing on a single channel.
CMD_RESET	Reset one or more channels their initialized state. All active and queued requests will be aborted.
CMD_START	Resume writes to one or more channels that were stopped.
CMD_STOP	Stop any write cycle in progress on one or more channels.
CMD_WRITE	Start a write cycle on a single channel.

Exec Functions as Used in This Chapter

-----	-----
AbortIO()	Abort a command to the audio device. If in progress, it is stopped immediately, otherwise it is removed from the queue.
BeginIO()	Initiate a command and return immediately (asynchronous request).

CheckIO()	Determine the current state of an I/O request.
CloseDevice()	Relinquish use of the audio device.
OpenDevice()	Obtain use of the audio device.
Wait()	Wait for a signal from the audio device.
WaitPort()	Wait for the audio message port to receive a message.

Exec Support Functions as Used in This Chapter

AllocMem()	Allocate a block of memory.
CreatePort()	Create a signal message port for reply messages from the audio device. Exec will signal a task when a message arrives at the reply port.
DeletePort()	Delete the message port created by CreatePort().
FreeMem()	Free a block of previously allocated memory.

1.5 2 Audio Device / Device Interface

The audio device operates like the other Amiga I/O devices. To make sound, you first open the audio device, then send I/O requests to it, and then close it when finished. See "Introduction to Amiga System Devices" chapter for general information on device usage.

Audio device commands use an extended I/O request block named IOAudio to send commands to the audio device. This is the standard IORequest block with some extra fields added at the end.

```
struct IOAudio
{
    struct IORequest ioa_Request; /* I/O request block. See exec/io.h. */
    WORD    ioa_AllocKey;        /* Alloc. key filled in by audio device */
    UBYTE   *ioa_Data;           /* Pointer to a sample or allocation mask */
    ULONG   ioa_Length;          /* Length of sample or allocation mask. */
    UWORD   ioa_Period;           /* Sample playback speed */
    UWORD   ioa_Volume;           /* Volume of sound */
    UWORD   ioa_Cycles;           /* # of times to play sample. 0=forever. */
    struct Message ioa_WriteMsg; /* Filled in by device- usually not used */
};
```

See the include file devices/audio.h for the complete structure definition.

Opening The Audio Device	Audio And System I/O Functions
Audio Device Command Types	Closing The Audio Device
Scope Of Audio Commands	

1.6 2 / Device Interface / Opening The Audio Device

Before you can use the audio device, you must first open it with a call to `OpenDevice()`. Four primary steps are required to open the audio device:

- * Create a message port using `CreatePort`. Reply messages from the device must be directed to a message port.
- * Allocate memory for an extended I/O request structure of type `IOAudio` using `AllocMem()`.
- * Fill in `io_Message.mn_ReplyPort` with the message port created by `CreatePort`.
- * Open the audio device. Call `OpenDevice()`, passing `IOAudio`.

```
struct MsgPort *AudioMP;          /* Define storage for port pointer */
struct IOAudio *AudioIO;          /* Define storage for IORequest pointer */

if (AudioMP = CreatePort(0,0) )
{
    AudioIO = (struct IOAudio *)
        AllocMem(sizeof(struct IOAudio), MEMF_PUBLIC | MEMF_CLEAR);
    if (AudioIO)
    {
        AudioIO->ioa_Request.io_Message.mn_ReplyPort = AudioMP;
        AudioIO->ioa_AllocKey = 0;
    }

    if (OpenDevice(AUDIONAME, 0L, (struct IORequest *)AudioIO, 0L) )
        printf("%s did not open\n", AUDIONAME);
}
```

A special feature of the `OpenDevice()` function with the audio device allows you to automatically allocate channels for your program to use when the device is opened. This is convenient since you must allocate one or more channels before you can produce sound.

This is done by setting `ioa_AllocKey` to zero, setting `ioa_Request.io_Message.mn_Node.ln_Pri` to the appropriate precedence, setting `io_Data` to the address of a channel combination array, and setting `ioa_Request.ioa_Length` to a non-zero value (the length of the channel combination array). The audio device will attempt to allocate channels just as if you had sent the `ADCMD_ALLOCATE` command (see below). If the allocation fails, the `OpenDevice()` call will return immediately.

If you want to allocate channels at some later time, set the `ioa_Request.ioa_Length` field of the `IOAudio` block to zero when you call `OpenDevice()`. For more on channel allocation and the `ADCMD_ALLOCATE` command, see the section on Allocation and Arbitration below.

```
UBYTE chans[] = {1,2,4,8}; /* get any of the four channels */

if (AudioIO)
{
    AudioIO->ioa_Request.io_Message.mn_ReplyPort = AudioMP;
}
```

```

AudioIO->ioa_AllocKey          = 0;
AudioIO->ioa_Request.io_Message.mn_Node.ln_Pri= 120;
AudioIO->ioa_Data              = chans;
AudioIO->ioa_Length            = sizeof(chans);
}

if (OpenDevice(AUDIONAME, 0L, (struct IORequest *)AudioIO, 0L) )
    printf("%s did not open\n", AUDIONAME);

```

1.7 2 / Device Interface / Audio Device Command Types

Commands for audio use can be divided into two categories: allocation/arbitration commands and hardware control commands.

There are four allocation/arbitration commands. These do not actually produce any sound. Instead they manage and arbitrate the audio resources for the many tasks that may be using audio in the Amiga's multitasking environment.

ADCMD_ALLOCATE	- Reserves an audio channel for your program to use.
ADCMD_FREE	- Frees an audio channel.
ADCMD_SETPREC	- Changes the precedence of a sound in progress.
ADCMD_LOCK	- Tells if a channel has been stolen from you.

The hardware control commands are used to set up, start, and stop sounds on the audio device:

CMD_WRITE	- The main command. Starts a sound playing.
ADCMD_FINISH	- Aborts a sound in progress.
ADCMD_PERVOL	- Changes the period (speed) and volume of a sound in progress.
CMD_FLUSH	- Clears the audio channels.
CMD_RESET	- Resets and initializes the audio device.
ADCMD_WAITCYCLE	- Signals you when a cycle finishes.
CMD_STOP	- Temporarily stops a channel from playing.
CMD_START	- Restarts an audio channel that was stopped.
CMD_READ	- Returns a pointer to the current IOAudio request.

1.8 2 / Device Interface / Scope Of Audio Commands

Most audio commands can operate on multiple channels. The exceptions are ADCMD_WAITCYCLE, CMD_WRITE and CMD_READ, which can only operate on one channel at a time. You specify the channel(s) that you want to use by setting the appropriate bits in the `ioa_Request.io_Unit` field of the IOAudio block. If you send a command for a channel that you do not own, your command will be ignored. For more details, see the section on "Allocation and Arbitration" below.

1.9 2 / Device Interface / Audio And System I/O Functions

```
BeginIO()  
Wait() and WaitPort()  
AbortIO()
```

1.10 2 / Audio And System I/O Functions / BeginIO()

All the commands that you can give to the audio device should be sent by calling the `BeginIO()` function. This differs from other Amiga devices which generally use `SendIO()` or `DoIO()`. You should not use `SendIO()` or `DoIO()` with the audio device because these functions clear some special flags used by the audio device; this might cause audio to work incorrectly under certain circumstances. To be safe, you should always use `BeginIO()` with the audio device.

1.11 2 / Audio And System I/O Functions / Wait() and WaitPort()

These functions can be used to put your task to sleep while a sound plays. `Wait()` takes a wake-up mask as its argument. The wake-up mask is usually the `mp_SigBit` of a `MsgPort` that you have set up to get replies back from the audio device.

`WaitPort()` will put your task to sleep while a sound plays. The argument to `WaitPort()` is a pointer to a `MsgPort` that you have set up to get replies back from the audio device.

`Wait()` and `WaitPort()` will not remove the message from the reply port. You must use `GetMsg()` to remove it.

You must always use `Wait()` or `WaitPort()` to wait for I/O to finish with the audio device.

1.12 2 / Audio And System I/O Functions / AbortIO()

This function can be used to cancel requests for `ADCMD_ALLOCATE`, `ADCMD_LOCK`, `CMD_WRITE`, or `ADCMD_WAITCYCLE`. When used with the audio device, `AbortIO()` always succeeds.

1.13 2 / Device Interface / Closing The Audio Device

An `OpenDevice()` must eventually be matched by a call to `CloseDevice()`.

All I/O requests must be complete before `CloseDevice()`. If any requests are still pending, abort them with `AbortIO()` :

```
AbortIO((struct IORequest *)AudioIO); /* Abort any pending requests */  
WaitPort(AudioMP);                  /* Wait for abort message */  
GetMsg(AudioMP);                     /* Get abort message */
```

```
CloseDevice((struct IORequest *)AudioIO);
```

CloseDevice() performs an ADCMD_FREE command on any channels selected by the ioa_Request.io_Unit field of the IOAudio request. This means that if you close the device with the same IOAudio block that you used to allocate your channels (or a copy of it), the channels will be automatically freed.

If you allocated channels with multiple allocation commands, you cannot use this function to close all of them at once. Instead, you will have to issue one ADCMD_FREE command for each allocation that you made. After issuing the ADCMD_FREE commands for each of the allocations, you can call CloseDevice().

1.14 2 Audio Device / A Simple Audio Example

The Amiga's audio software has a complex allocation and arbitration system which is described in detail in the sections below. At this point, though, it may be helpful to look at a simple audio example:

```
Audio.c
```

1.15 2 Audio Device / Audio Allocation and Arbitration

The first command you send to the audio device should always be ADCMD_ALLOCATE. You can do this when you open the device, or at a later time. You specify the channels you want in the ioa_Data field of the IOAudio block. If the allocation succeeds, the audio device will return the channels that you now own in the lower four bits of the ioa_Request.io_Unit field of your IOAudio block. For instance, if the io_Unit field equals 5 (binary 0101) then you own channels 2 and 0. If the io_Unit field equals 15 (binary 1111) then you own all the channels.

When you send the ADCMD_ALLOCATE command, the audio device will also return a unique allocation key in the ioa_AllocKey of the IOAudio block. You must use this allocation key for all subsequent commands that you send to the audio device. The audio device uses this unique key to identify which task issued the command. If you do not use the correct allocation key assigned to you by the audio device when you send a command, your command will be ignored.

When you request a channel with ADCMD_ALLOCATE, you specify a precedence number from -128 to 127 in the ioa_Request.io_Message.mn_Node.ln_Pri field of the IOAudio block. If a channel you want is being used and you have specified a higher precedence than the current user, ADCMD_ALLOCATE will "steal" the channel from the other user. Later on, if your precedence is lower than that of another user who is performing an allocation, the channel may be stolen from you.

If you set the precedence to 127 when you open the device or raise the precedence to 127 with the ADCMD_SETPREC command, no other tasks can steal a channel from you. When you have finished with a channel, you must relinquish it with the ADCMD_FREE command to make it available for other

users.

The following table shows suggested precedence values.

SUGGESTED PRECEDENCES FOR CHANNEL ALLOCATION

Precedence -----	Type of Sound -----
127	Unstoppable. Sounds first allocated at lower precedence, then set to this to the highest level.
90 - 100	Emergencies. Alert, urgent situation that requires immediate action.
80 - 90	Annunciators. Attention, bell (CTRL-G).
75	Speech. Synthesized or recorded speech (narrator.device).
50 - 70	Sonic cues. Sounds that provide information that is not provided by graphics. Only the beginning of each sound (enough to recognize it) should be at this level; the rest should be set to sound effects level.
-50 - 50	Music program. Musical notes in music-oriented program. The higher levels should be used for the attack portions of each note.
-70 - -50	Sound effects. Sounds used in conjunction with graphics. More important sounds should use higher levels.
-100 - -80	Background. Theme music and restartable background sounds.
-128	Silence. Lowest level (freeing the channel completely is preferred).

If you attempt to perform a command on a channel that has been stolen from you by a higher priority task, an `AUDIO_NOALLOCATION` error is returned and the bit in the `ioa_Request.io_Unit` field corresponding to the stolen channel is cleared so you know which channel was stolen.

If you want to be warned before a channel is stolen so that you have a chance to stop your sound gracefully, then you should use the `ADCMD_LOCK` command after you open the device. This command is also useful for programs which write directly to the audio hardware. For more on `ADCMD_LOCK`, see the section below.

1.16 2 Audio Device / Allocation and Arbitration Commands

These commands allow the audio channels to be shared among different tasks and programs. None of these commands can be called from interrupt code.

`ADCMD_ALLOCATE` `ADCMD_FREE` `ADCMD_SETPREC` `ADCMD_LOCK`

1.17 2 / Allocation and Arbitration Commands / ADCMD_ALLOCATE

This command gives your program a channel to use and should be the first command you send to the audio device. You specify the channels you want by setting a pointer to an array in the `ioa_Data` field of the `IOAudio` structure. This array uses a value of 1 to allocate channel 0, 2 for channel 1, 4 for channel 2, and 8 for channel 3. For multiple channels, add the values together. For example, if you want to allocate all channels, use a value of 15.

If you want a pair of stereo channels and you have no preference about which of the left and right channels the system will choose for the allocation, you can pass a pointer to an array containing 3, 5, 10, and 12. Channels 1 and 2 produce sound on the left side and channels 0 and 3 on the right side. The table below shows how this array corresponds to all the possible combinations of a right and a left channel.

POSSIBLE CHANNEL COMBINATIONS				
Channel 3 (right)	Channel 2 (left)	Channel 1 (left)	Channel 0 (right)	Decimal Value of Allocation Mask
-----	-----	-----	-----	-----
0	0	1	1	3
0	1	0	1	5
1	0	1	0	10
1	1	0	0	12

How ADCMD_ALLOCATE Operates
The ADIOF_NOWAIT Flag

ADCMD_ALLOCATE Examples
The Allocation Key

1.18 2 / ADCMD_ALLOCATE / How ADCMD_ALLOCATE Operates

The `ADCMD_ALLOCATE` command tries the first combination, 3, to see if channels 0 and 1 are not being used. If they are available, the 3 is copied into the `io_Unit` field and you get an allocation key for these channels in the `ioa_AllocKey` field. You copy the key into other I/O blocks for any other commands you may want to perform on these channels.

If channels 0 and 1 are being used, `ADCMD_ALLOCATE` tries the other combinations in turn. If all the combinations are in use, `ADCMD_ALLOCATE` checks the precedence number of the users of the channels and finds the combination that requires it to steal the channel or channels of the lowest precedence. If all the combinations require stealing a channel or channels of equal or higher precedence, the `ADCMD_ALLOCATE` request fails. Precedence is in the `ln_Pri` field of the `io_Message` in the `IOAudio` block you pass to `ADCMD_ALLOCATE`; it has a value from -128 to 127.

1.19 2 / ADCMD_ALLOCATE / The ADIOF_NOWAIT Flag

If you need to produce a sound right now and otherwise don't want to allocate, set the `ADIOF_NOWAIT` flag to 1. This will cause the command to return an `IOERR_ALLOCFAILED` error if it cannot allocate any of the

channels. If you are producing a non-urgent sound and you can wait, set the ADIOF_NOWAIT flag to 0. Then, the IOAudio block returns only when you get the allocation. If ADIOF_NOWAIT is set to 0, the audio device will continue to retry the allocation request whenever channels are freed until it is successful. If the program decides to cancel the request, AbortIO() can be used.

1.20 2 / ADCMD_ALLOCATE / ADCMD_ALLOCATE Examples

The following are more examples of how to tell ADCMD_ALLOCATE your channel preferences. If you want any channel, but want a right channel first, use an array containing 1, 8, 2, and 4:

```
0001
1000
0010
0100
```

If you only want a right channel, use 1 and 8 (channels 0 and 3):

```
0001
1000
```

If you want only a left channel, use 2 and 4 (channels 1 and 2):

```
0010
0100
```

If you want to allocate a channel and keep it for a sound that can be interrupted and restarted, allocate it at a certain precedence. If it is stolen, allocate it again with the ADIOF_NOWAIT flag set to 0. When the channel is relinquished, you will get it again.

1.21 2 / ADCMD_ALLOCATE / The Allocation Key

If you want to perform multi-channel commands, all the channels must have the same key since the IOAudio block has only one allocation key field. The channels must all have that same key even when they were not allocated simultaneously. If you want to use a key you already have, you can pass that key in the ioa_AllocKey field and ADCMD_ALLOCATE can allocate other channels with that existing key. The ADCMD_ALLOCATE command returns a new and unique key only if you pass it a zero in the allocation key field.

1.22 2 / Allocation and Arbitration Commands / ADCMD_FREE

ADCMD_FREE is the opposite of ADCMD_ALLOCATE. When you perform ADCMD_FREE on a channel, it does a CMD_RESET command on the hardware and "unlocks" the channel. It also checks to see if there are other pending allocation requests. You do not need to perform ADCMD_FREE on channels stolen from you. If you want channels back after they have been stolen, you must

reallocate them with the same allocation key.

1.23 2 / Allocation and Arbitration Commands / ADCMD_SETPREC

This command changes the precedence of an allocated channel. As an example of the use of ADCMD_SETPREC, assume that you are making the sound of a chime that takes a long time to decay. It is important that user hears the chime but not so important that he hears it decay all the way. You could lower precedence after the initial attack portion of the sound to let another program steal the channel. You can also set the precedence to maximum (127) if you do not want the channel(s) stolen from you.

1.24 2 / Allocation and Arbitration Commands / ADCMD_LOCK

The ADCMD_LOCK command performs the "steal verify" function. When another application is attempting to steal a channel or channels, ADCMD_LOCK gives you a chance to clean up before the channel is stolen. You perform a ADCMD_LOCK command right after the ADCMD_ALLOCATE command. ADCMD_LOCK does not return until a higher-priority user attempts to steal the channel(s) or you perform an ADCMD_FREE command. If someone is attempting to steal, you must finish up and ADCMD_FREE the channel as quickly as possible.

You must use ADCMD_LOCK if you want to write directly to the hardware registers instead of using the device commands. If your channel is stolen, you are not notified unless the ADCMD_LOCK command is present. This could cause problems for the task that has stolen the channel and is now using it at the same time as your task. ADCMD_LOCK sets a switch that is not cleared until you perform an ADCMD_FREE command on the channel. Canceling an ADCMD_LOCK request with AbortIO() will not free the channel.

The following outline describes how ADCMD_LOCK works when a channel is stolen and when it is not stolen.

1. User A allocates a channel.
2. User A locks the channel.

If User B allocates the channel with a higher precedence:

3. User B's ADCMD_ALLOCATE command is suspended (regardless of the setting of the ADIOF_NOWAIT flag).
4. User A's lock command is replied to with an error (ADIOERR_CHANNELSTOLEN).
5. User A does whatever is needed to finish up when a channel is stolen.
6. User A frees the channel with ADCMD_FREE.
7. User B's ADCMD_ALLOCATE command is replied to. Now user B has the channel.

Otherwise, if the channel is not allocated by another user:

3. User A finishes the sound.
 4. User A performs the ADCMD_FREE command.
-

5. User A's ADCMD_LOCK command is replied to.

Never make the freeing of a channel (if the channel is stolen) dependent on allocating another channel. This may cause a deadlock. If you want channels back after they have been stolen, you must reallocate them with the same allocation key. To keep a channel and never let it be stolen, set precedence to maximum (127). Do not use a lock for this purpose.

1.25 2 Audio Device / Hardware Control Commands

The following commands change hardware registers and affect the actual sound output.

CMD_WRITE	CMD_FLUSH	CMD_STOP
ADCMD_FINISH	CMD_RESET	CMD_START
ADCMD_PERVOL	ADCMD_WAITCYCLE	CMD_READ

1.26 2 / Hardware Control Commands / CMD_WRITE

This is a single-channel command and is the main command for making sounds. You pass the following to CMD_WRITE:

- * A pointer to the waveform to be played (must start on a word boundary and must be in memory accessible by the custom chips, MEMF_CHIP)
- * The length of the waveform in bytes (must be an even number)
- * A count of how many times you want to play the waveform

If the count is 0, CMD_WRITE will play the waveform from beginning to end, then repeat the waveform continuously until something aborts it.

If you want period and volume to be set at the start of the sound, set the WRITE command's ADIOF_PERVOL flag. If you do not do this, the previous volume and period for that channel will be used. This is one of the flags that is cleared by DoIO() and SendIO(). The ioa_WriteMsg field in the IOAudio block is an extra message field that can be replied to at the start of the CMD_WRITE. This second message is used only to tell you when the CMD_WRITE command starts processing, and it is used only when the ADIOF_WRITEMESSAGE flag is set to 1.

If a CMD_STOP has been performed, the CMD_WRITE requests are queued up. The CMD_WRITE command does not make its own copy of the waveform, so any modification of the waveform before the CMD_WRITE command is finished may affect the sound. This is sometimes desirable for special effects. To splice together two waveforms without clicks or pops, you must send a separate, second CMD_WRITE command while the first is still in progress. This technique is used in double-buffering, which is described below.

By using two waveform buffers and two CMD_WRITE requests you can compute a waveform continuously. This is called double-buffering. The following describes how you use double-buffering.

1. Compute a waveform in memory buffer A.
2. Issue CMD_WRITE A with io_Data pointing to buffer A.
3. Continue the waveform in memory buffer B.
4. Issue CMD_WRITE B with io_Data pointing to Buffer B.
5. Wait for CMD_WRITE A to finish.
6. Continue the waveform in memory buffer A.
7. Issue CMD_WRITE A with io_Data pointing to Buffer A.
8. Wait for CMD_WRITE B to finish.
9. Loop back to step 3 until the waveform is finished.
10. At the end, remember to wait until both CMD_WRITE A and B are finished.

1.27 2 / Hardware Control Commands / ADCMD_FINISH

The ADCMD_FINISH command aborts (calls AbortIO()) the current write request on a channel or channels. This is useful if you have something playing, such as a long buffer or some repetitions of a buffer, and you want to stop it.

ADCMD_FINISH has a flag you can set (ADIOF_SYNCCYCLE) that allows the waveform to finish the current cycle before aborting it. This is useful for splicing together sounds at zero crossings or some other place in the waveform where the amplitude at the end of one waveform matches the amplitude at the beginning of the next. Zero crossings are positions within the waveform at which the amplitude is zero. Splicing at zero crossings gives you fewer clicks and pops when the audio channel is turned off or the volume is changed.

1.28 2 / Hardware Control Commands / ADCMD_PERVOL

ADCMD_PERVOL lets you change the volume and period of a CMD_WRITE that is in progress. The change can take place immediately or you can set the ADIOF_SYNCCYCLE flag to have the change occur at the end of the cycle. This is useful to produce vibratos, glissandos, tremolos, and volume envelopes in music or to change the volume of a sound.

1.29 2 / Hardware Control Commands / CMD_FLUSH

CMD_FLUSH aborts (calls AbortIO()) all CMD_WRITE and all ADCMD_WAITCYCLES that are queued up for the channel or channels. It does not abort ADCMD_LOCKs (only ADCMD_FREE clears locks).

1.30 2 / Hardware Control Commands / CMD_RESET

CMD_RESET restores all the audio hardware registers. It clears the attach bits, restores the audio interrupt vectors if the programmer has changed them, and performs the CMD_FLUSH command to cancel all requests to the channels. CMD_RESET also unstops channels that have had a CMD_STOP performed on them. CMD_RESET does not unlock channels that have been locked by ADCMD_LOCK.

1.31 2 / Hardware Control Commands / ADCMD_WAITCYCLE

This is a single-channel command. ADCMD_WAITCYCLE is replied to when the current cycle has completed. If there is no CMD_WRITE in progress, it returns immediately.

1.32 2 / Hardware Control Commands / CMD_STOP

This command stops the current write cycle immediately. If there are no CMD_WRITES in progress, it sets a flag so any future CMD_WRITES are queued up and do not begin processing (playing).

1.33 2 / Hardware Control Commands / CMD_START

CMD_START undoes the CMD_STOP command. Any cycles that were stopped by the CMD_STOP command are actually lost because of the impossibility of determining exactly where the DMA ceased. If the CMD_WRITE command was playing two cycles and the first one was playing when CMD_STOP was issued, the first one is lost and the second one will be played.

This command is also useful when you are playing the same wave form with the same period out of multiple channels. If the channels are stopped when the CMD_WRITE commands are issued, CMD_START exactly synchronizes them, avoiding cancellation and distortion. When channels are allocated, they are effectively started by the CMD_START command.

1.34 2 / Hardware Control Commands / CMD_READ

CMD_READ is a single-channel command. Its only function is to return a pointer to the current CMD_WRITE command. It enables you to determine which request is being processed.

1.35 2 Audio Device / Double Buffered Sound Example

The program listed below demonstrates double buffering with the audio device. Run the program from the CLI. It takes one parameter – the name of an IFF 8SVX sample file to play on the Amiga's audio device. The maximum size for a sample on the Amiga is 128K. However, by using double-buffering and queueing up requests to the audio device, you can play longer samples smoothly and without breaks.

Audio_8SVX.c

1.36 2 Audio Device / Additional Information on the Audio Device

Additional programming information on the audio device can be found in the include files and the Autodocs for the audio device. Both are contained in the Amiga ROM Kernel Reference Manual: Includes and Autodocs. Information can also be found in the Amiga Hardware Reference Manual.

Audio Device Information	

INCLUDES	devices/audio.h devices/audio.i
AUTODOCS	audio.doc