Sound Driver

Version 1.2.0

The contents in this document are highly confidential and should be handled accordingly.

NTR-06-0362-001-A2
Released: February 2, 2006

Contents

NTR-06-0362-001-A2
Released: February 2, 2006

Code

Tables

Figures

Revision History

Version	Revision Date	Description
1.2.0	7/4/2005	• Revised explanation of commands Added the term "command packet" and unified other terminology • Added a section about the problem with Sound Capture. • Fixed errors in code 7-1
1.1.1	5/10/2005	• Corrected the description of command states • Corrected the explanation of command tags • Fixed writing errors
1.1.0	4/26/2005	• Fixed Sound Capture writing errors • Made revisions in line with addition to sound functions • Corrected the explanation of commands
1.0.0	4/13/2005	Initial version

1 Introduction

1.1 Overview

Sound Driver (SND) is a library that gives applications some relatively low-level control over the Nintendo DS sound hardware. The Nintendo DS sound features can be used with this library implemented on the ARM7.

This document describes the mechanism of Sound Driver operation and explains its essential group of functions. For detailed explanations about specific functions, see the Function Reference.

1.2 Structure of Sound Driver

The sound driver can be broadly divided into three library parts:

1. ARM9 library that provides the library's interface.
2. The core part of the library on the ARM7.
3. The command library in charge of data exchanges between the ARM9 and the ARM7.

Sound Driver is used via the group of functions on the ARM9, but you should still understand the flow of operations from the time the functions are called to the time the sounds are actually processed on the ARM7. Of particular importance is an understanding of the exchange of commands between the ARM9 and the ARM7. This is covered in detail in Chapter 3, The ARM7 Command Process.

1.3 NITRO-Composer

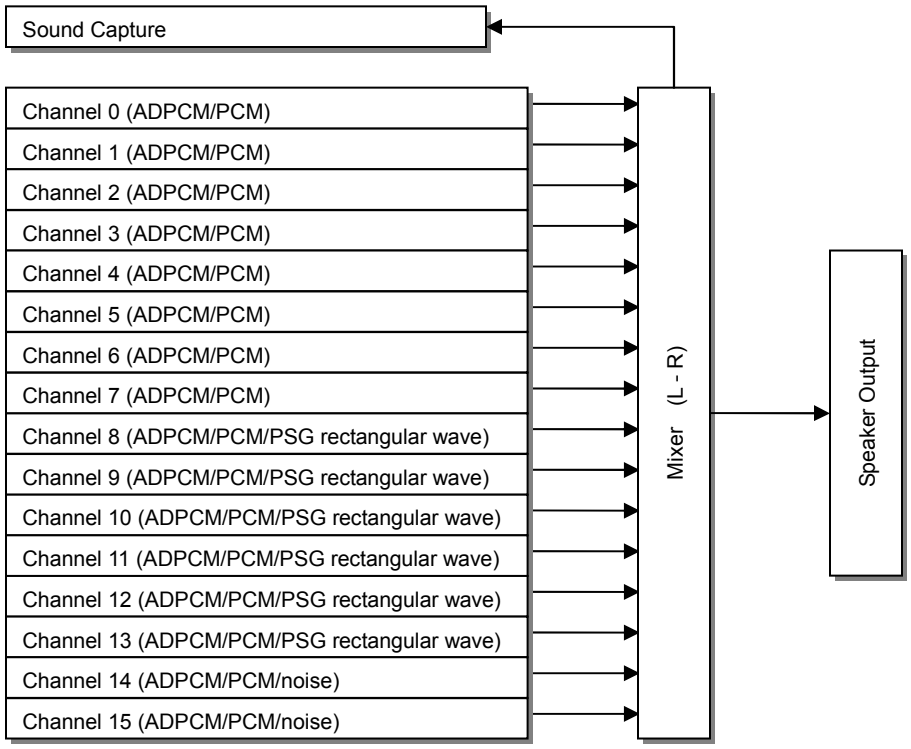
The NITRO-System package comes with a sound library called NITRO-Composer that can be used for the playback of sequences and streaming, and for the management of sound data. NITRO-Composer allows it to be used with Sound Driver's group of low-level functions, but certain precautions should be considered; These precautions are explained in Chapter 8, Precautions About Use with NITRO-Composer. Since NITRO-Composer and Sound Driver share the same formats for sequence or bank data, the documentation for NITRO-Composer should also be reviewed.

2 The Sound Hardware

2.1 Overview of the Sound Circuitry

The Nintendo DS sound hardware consists of 16 sets of sound circuitry that independently control 16 channels, a mixer to blend the sounds from the separate circuits, and a sound capture component that writes the sound output to memory.

Figure 2-1 Schematic of the Sound Circuitry



2.2 Channels

Each sound circuit is called a channel. Each channel can generate one sound. Therefore, the 16 channels can play up to 16 sounds. The channels are numbered channel 0 to channel 15. As shown in Table 2-1, each channel number has different capabilities.

Table 2-1 The Channel Numbers and Their Features

Channel Numbers	Features
0, 2	These channels can play ADPCM/PCM. In addition, the output from these channels can serve as the input for sound capture.
1, 3	These channels can play ADPCM/PCM. Sound Capture shares timers with these channels, so when sound capture is being used these channels can only be used as the output channels for sound capture.
4 to 7	These channels can play ADPCM/PCM.
8 to 13	These channels can play ADPCM/PCM as well as PSG rectangular waves.
14, 15	These channels can play ADPCM/PCM as well as white noise.

2.2.1 ADPCM/PCM

Channels that play ADPCM/PCM can play 16-bit PCM, 8-bit PCM, and IMA-ADPCM.

2.2.2 PSG Rectangular Waves

Channels that play PSG rectangular waves can play rectangular waves for which the duty ratio can be set.

2.2.3 Noise

Channels that play noise can play white noise. There are no configuration settings for white noise.

2.3 Sound Capture

The Nintendo DS has two built-in sound capture components for writing output waveform data to memory. Figure 2.1 depicts the capture of the left output and the right output from the mixer. Channel 0 and 2 can be used to capture sound also.

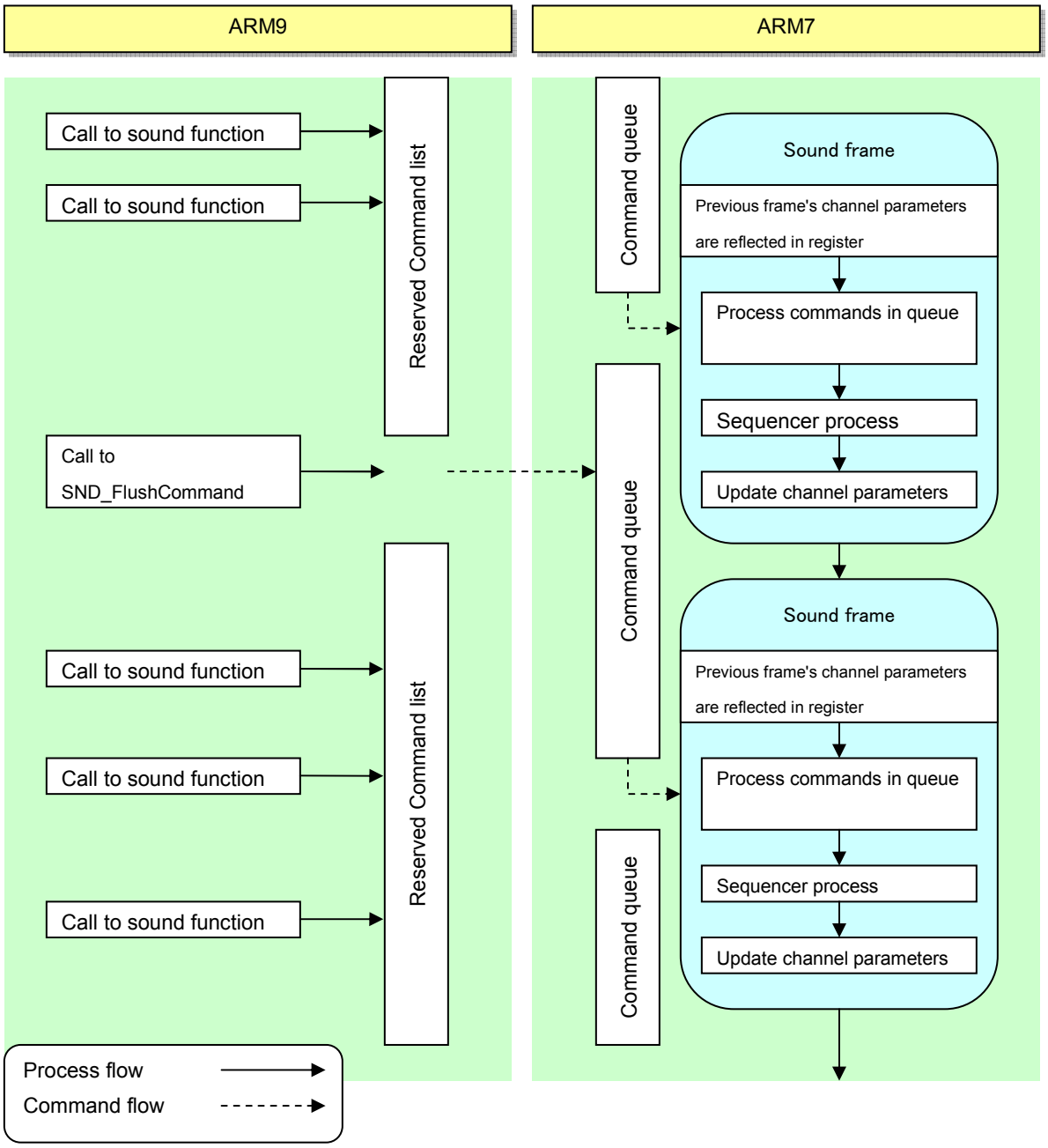
Resolution of the captured waveform can be set to either 8-bit or 16-bit.

3 The ARM7 Command Process

3.1 The Command Process Flow

When SND functions are called, processes do not necessarily begin immediately. SND functions are first added to the ARM9 reserved command list, After the `SND_FlushCommand` is called, the ARM7 begins to process the commands in the ARM9 reserved command list.

Figure 3-1 The Command Process Flow



3.2 Sound Functions and Commands

Most of the SND library functions are commands for performing processes on the ARM7. Commands are stored in ARM9 and sent to ARM7 for process when `SND_FlushCommand` is explicitly executed.

Some of the functions that do not require processing on the ARM7 are executed when they are called.

The following functions are processed on the ARM7:

Sequence Commands

```
SND_StartSeq                        SND_PrepareSeq
SND_StartPreparedSeq                SND_StopSeq
SND_PauseSeq                        SND_SetPlayerTempoRatio
SND_SetPlayerVolume                 SND_SetPlayerChannelPriority
SND_SetPlayerLocalVariable          SND_SetPlayerGlobalVariable
SND_SetTrackMute                    SND_SetTrackVolume
SND_SetTrackPitch                   SND_SetTrackPan
SND_SetTrackModDepth                SND_SetTrackModSpeed
SND_SetTrackAllocatableChannel
```

Channel Commands

```
SND_LockChannel                     SND_UnlockChannel
SND_StopUnlockedChannel             SND_SetupChannelPcm
SND_SetupChannelPsg                 SND_SetupChannelNoise
SND_SetChannelVolume                SND_SetChannelTimer
SND_SetChannelPan
```

Capture Commands

```
SND_SetupCapture
```

Alarm Commands

```
SND_SetupAlarm
```

Timer Commands

```
SND_StartTimer                      SND_StopTimer
```

Global Settings Commands

```
SND_SetMasterVolume                 SND_SetMasterPan
SND_ResetMasterPan                  SND_SetOutputSelector
```

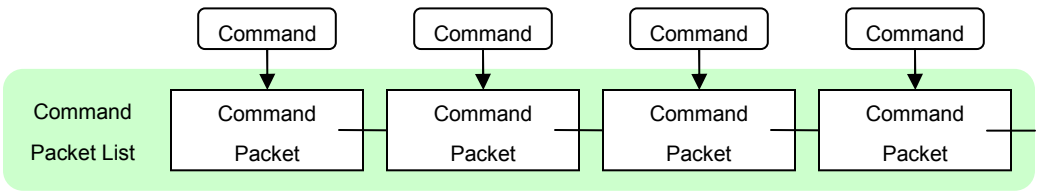
Data Invalidation Commands

```
SND_InvalidateSeqData               SND_InvalidateBankData
SND_InvalidateWaveData
```

3.3 The Command Packets

Command packets have been prepared for the mechanism that sends commands to ARM7. Each command packet contains a single command. These command packets are bundled into in a command packet list that sends the commands to the ARM7.

Figure 3-2 Command packet



Each command packet can be in any of the following states:

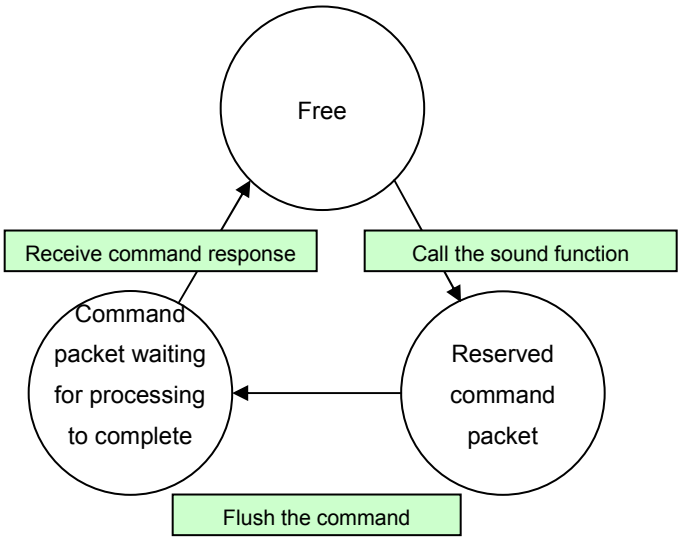
1. The "free" state–the command can be newly registered.
2. The "reserved" state–the command has been registered and is waiting to be flushed.
3. The "wait" state–the command waits for the process to complete in the ARM7.

For a command in the reserved state to execute, the reserved command packet list must be flushed. This operation is called a command flush. If a command packet is waiting for processes to end, the command packet cannot return to the free state until a command response confirms that the process has completed.

You are limited to a maximum of 256 command packets. Because of this limit on command packets, you need to periodically flush commands, receive command responses, and secure free command packets.

You can use the following functions that return the number of command packets in each of the three states: `SND_CountFreeCommand`, `SND_CountReservedCommand`, and `SND_CountWaitingCommand`.

Figure 3-2 Command Packet State Transitions



3.4 Flushing Command

`SND_FlushCommand` can be used when necessary. If the call needs to be synchronous with ARM7 processes, you can combine `SND_FlushCommand` with a command tag (see 3.6). Commands will not be processed if they are not flushed. If you do not periodically flush the reserved command packet list, the list will grow and you will have a shortage of free command packets. Calling `SND_FlushCommand` once every frame is recommended.

3.5 Receiving Command Response

A command packet that is waiting for processes to complete is not free until a response is received that indicates the process of the flushed command has ended. By calling SND_RecvCommandReply, you can take the oldest processed and completed command to the free packet list and make the command packet list free and get that list.

As with receiving command response, if you do not periodically call SND_RecvCommandReply, you will have a shortage of free command packets. To ensure enough free command packets, you should periodically call SND_RecvCommandReply.

The following code shows a process where the sound function SoundMain is called in every frame. The flushing command and receiving command response is called in every frame.

Code 3-1 The Command Flush and Command Response Processes

```
void SoundMain( void )
{
    // Receive ARM7 response
    while ( SND_RecvCommandReply( SND_COMMAND_NOBLOCK ) != NULL ) {}

    // Issue command to ARM7
    SND_FlushCommand( SND_COMMAND_NOBLOCK );
}
```

SND_FlushCommand and SND_RecvCommandReply each have a parameter that can be used to specify whether to block inside the function until the process succeeds. In the example above, the parameter for SND_RecvCommandReply is set to SND_COMMAND_NOBLOCK so that no block is performed inside the function. You should specify SND_COMMAND_BLOCK in these functions if you need to be certain that the flushing of commands and the reception of command responses has completed successfully.

3.6 Command Tags

Command tags can be used to determine whether or not the processing of commands has finished, and also to synchronize the command processing in the ARM7 with the application in the ARM9.

Call the SND_GetCurrentCommandTag function to get a command tag and check whether the commands prior to the acquisition of the tag have finished executing. Use SND_IsFinishedCommandTag to check whether commands prior to the tag specified in the argument have finished executing. The function SND_WaitForCommandProc asks the ARM7 to quickly execute any commands prior to the acquisition of the tags that have not finished executing; processing waits inside this function until execution has been completed.

A flush command must be performed before processing is complete, so be careful when using command tags to check if a process has finished.

3.7 When There is a Shortage of Free Command Packets

If there is a shortage of free command packets, new commands cannot be added unless something is done about the situation. If a sound function is called as part of a new command process when there is a shortage of free command packets, the following procedure is performed in order to secure enough free command packets:

1. If a command is in the wait state, act to receive a command response.
2. If (1) does not solve the shortage, flush the command, request immediate execution by ARM7, and wait until there is a command response.

Because this procedure includes both a command flush and a wait for a command response, in some cases it may take some time before the sound function is called. Furthermore, if the flush is done while they are executing, the command flush might separate processes that should be executing simultaneously. To prevent processes from being separated, there is the `SND_WaitForFreeCommand` function, which waits until a specified number of free commands have been secured. To avoid the problem of insufficient free command packets, you should periodically perform the processes to flush commands and receive command responses.

3.8 Sound Frames

The ARM7 sound frame interval is approximately 5.2 ms. Depending on the circumstances, it can take as long as the sound-frame interval for the process to execute after the command that has been flushed. The exception is when the argument `SND_COMMAND_IMMEDIATE` is specified when the command is flushed. When this exception occurs, the process can begin immediately in ARM7 without waiting for the next sound frame.

4 Playing Sounds

4.1 Playing Sequences and Controlling Channels

Sound Driver uses two methods to generate sounds. One involves the playback of performance sequences based on a special format of sequence data. The other involves the direct control of channels to make sounds.

Since NITRO-Composer offers more advanced processes for the performance of sequence data, here we will concentrate on an explanation of the direct control of channels to make sounds.

4.2 Controlling Channels

Generating sounds by directly controlling channels involves the following procedure:

1. Lock the channel
2. Set up the channel in accordance with the chosen playback method
3. Start the timer

4.2.1 Locking Channels

In order for the ARM7-implemented sequencer to automatically play sounds, channels are first reserved and later released after the sounds have finished playing. For this reason, when the programmer intends to directly control channels, the channels need to be locked by calling `SND_LockChannel` so the actions do not collide with those of the sequencer. Operations that are performed on channels are designed on the assumption that the channels have been locked.

A locked channel cannot be used by the sequencer, so when a locked channel is no longer needed, be sure to call `SND_UnlockChannel` and unlock the channel so it becomes available to the sequencer again.

4.2.2 Setting up Channels

Once channels have been locked, call the setup function that fits your purpose: `SND_SetupChannelPcm` for PCM playback, `SND_SetupChannelPsg` for PGS rectangular wave playback, and `SND_SetupChannelNoise` for white noise. In the case of PGS rectangular waves and white noise, only those channel numbers that can play those types of data can be set up.

4.2.3 Starting and Stopping the Timer

A channel begins to play sounds once `SND_StartTimer` is called and the timer has been started. Since one call to `SND_StartTimer` can start the time simultaneously for multiple channels, this is a way to coordinate the playing of sounds in multiple channels. The same timer-start call can be used to coordinate Sound Capture and Sound Alarm, both of which are described below.

To stop sounds, call `SND_StopTimer` to stop the timer. This can be used to simultaneously control multiple channels, just like the call to start the timer.

4.2.4 Channel Parameters

Volume, timer, and pan values can be set for each channel. The values can be set using the setup functions, but they can also be set individually by calling the `SND_SetChannelVolume`, `SND_SetChannelTimer`, and `SND_SetChannelPan` functions. This provides a way of changing the values even after the timer has started.

5 Sound Capture

5.1 Overview of Sound Capture

The Nintendo DS sound capture feature has two components.

Table 5-1 Capture Feature Components

Number	Capture target	Timer
Capture 0	Captures the output from the mixer's left channel or from channel 0.	Shares the channel 1 timer.
Capture 1	Captures the output from the mixer's right channel or from channel 2.	Shares the channel 3 timer.

Because the channel 1 and channel 3 timers are shared with sound capture, you lose the ability to freely set the timer values and generate sounds while using the capture feature. However, the captured data can still be re-output and used for other purposes.

5.2 How to Use Sound Capture

The procedure for using sound capture is similar to the procedure for using channels:

- Lock the channel that uses the timer being shared with the capture feature.
- Call `SND_SetupCapture` to set up the capture parameters.
- Call `SND_SetChannelTimer` to set the frequency of the shared timer, or call `SND_SetupChannelPcm` to configure the settings to play the captured data.
- Start the timer.

You can create sound effects by performing arithmetic processes on the captured data and then outputting the data again. NITRO-Composer makes use of sound capture to implement reverb and output effects.

5.3 Problems with Sound Capture

There is a problem with the Sound Capture hardware that prevents the correct capture of data when the output is being captured from channel 0 or channel 2.

For details about this problem, see the NITRO Programming Manual.

6 Sound Alarms

6.1 Overview of Sound Alarms

Sound Alarms is an alarm system that uses the ARM7 timer. You can use Sound Alarms to synchronize such process as the capture of sound data and the generation of sounds in channels. There are eight sound alarms, numbered 0 to 7, and all eight can be used at the same time.

6.2 How to Use Sound Alarms

The procedure for using Sound Alarms is shown below:

- Call `SND_SetupAlarm` to set up the sound alarms you plan to use.
- Call `SND_StartTimer` to start the configured sound alarm(s).

`SND_StartTimer` can start channels, sound capture, and sound alarms all at the same time. Start the sound alarms together with the channels and the sound captures you want to synchronize.

7 Getting Driver Information

7.1 Overview

You can learn information about the current driver state by using the group of functions that have been prepared to get this information. You just need to be careful about synchronizing with ARM9 when you act to get information that is being processed by ARM7.

7.2 Getting the Information Structure

You can get information about the current status of the channel, player, and track being processed by the ARM7. To synchronize these actions, use the following procedure:

- Call `SND_ReadDriverInfo` and get the driver information. This function is a command-reservation function, so in order to access the obtained information you must flush the command and wait for the command to finish executing.
- Call the pertinent functions to get the channel, player, and track information (`SND_ReadChannelInfo`, `SND_ReadPlayerInfo`, and `SND_ReadTrackInfo`).

The following example is code that gets the structure for the driver information, waits for the command to complete, and then gets other information.

Code 7-1 Getting the Driver Information Structure

```
u32 tag;
SNDDriverInfo driverInfo;
SNDChannelInfo channelInfo;
SNDPlayerInfo playerInfo;
SNDTrackInfo trackInfo;

/* Wait for completed obtainment of driver information */
SND_ReadDriverInfo( &driverInfo );
tag = SND_GetCurrentCommandTag( );
SND_FlushCommand( SND_COMMAND_BLOCK );
SND_WaitForCommandProc( tag );

/* Get information about channel 0 */
SND_ReadChannelInfo( &driverInfo, 0, &channelInfo );

/* Get information about player 1 */
SND_ReadPlayerInfo( &driverInfo, 1, &playerInfo );

/* Get information about track 3 of player 0 */
SND_ReadTrackInfo( &driverInfo, 0, 3, &trackInfo );
```

7.3 Getting Other Information

Besides these functions that get structure information, there are also some functions that be used to get information without the call to `SND_ReadDriverInfo`.

These functions are executed asynchronously from ARM7 command functions. As a result, you cannot be certain of the reason when one of the functions returns 0. For example, if after executing `SND_StartTimer` you were to call `SND_GetPlayerStatus` and the function returned 0, you could not determine whether the channel was not active because a command was not completed or because playback had ended.

In order to gain synchronization for the acquisition of information, use a command tag and call `SND_WaitForCommandProc` or devise some other means of checking to see whether the ARM7 command has completed processing.

Table 7-1 Other Functions

Function	Description
`SND_GetPlayerStatus`	Obtains the player status
`SND_GetChannelStatus`	Obtains the channel status
`SND_GetCaptureStatus`	Obtains the Sound Capture status
`SND_GetPlayerLocalVariable`	Obtains the sequence local variable
`SND_GetPlayerGlobalVariable`	Obtains the sequence global variable
`SND_GetPlayerTickCounter`	Obtains the sequence tick counter

8 Precautions About Use with NITRO-Composer

8.1 Using Player

When you use NITRO-Composer to play sequence data, you cannot control the player from Sound Driver. In other words, you cannot use NITRO-Composer sequence playback and Sound Driver sequence playback at the same time.

8.2 Using Channels

The Sound Driver functions for locking and unlocking channels (`SND_LockChannel` and `SND_UnlockChannel`) can only lock channels with regard to Sound Driver sequence playback. If you want to lock channels when using NITRO-Composer, use the NITRO-Composer functions `NNS_SndLockChannel` and `NNS_SndUnlockChannel`.

8.3 Using Sound Capture

If you want to use the Sound Driver's sound capture feature while using NITRO-Composer, call the NITRO-Composer function `NNS_SndLockCapture` so NITRO-Composer will not use the capture feature.

You will also need to call `NNS_SndLockChannel` at the same time to lock channel 1 and channel 3.

8.4 Using Sound Alarms

NITRO-Composer uses sound alarms internally, so if you are using NITRO-Composer and plan to use sound alarms yourself, call the NITRO-Composer function `NNS_SndAllocAlarm` to determine which alarm numbers are available. The numbers obtained by this function represent the alarms that are not being used internally by NITRO-Composer.

When you are done using an alarm, remember to release it by calling `NNS_SndFreeAlarm`.