Tapwave® Advanced Sound API Reference

Version 1.1a

Copyright

1. Application Sound Level

Tapwave devices have a master control for device-wide volume and muting, and APIs are provided to control these settings. The master volume control is referred to as the *primary* volume. We encourage developers to create applications that either use the *primary* volume or respect it by “mixing” their own sounds into other sounds the device may make. If an application plays a sound at 100%, it plays at the user’s preferred volume (or not at all if the volume is muted.) Applications can only reduce the primary volume, they cannot increase it to play sounds at “110%” of the user’s chosen volume.

For compatibility with non-Tapwave devices, applications can read (and attempt to set) the Game Sound Level in the system preferences, however on Tapwave devices this value is locked at 100%. This can be obtained by calling the Palm OS API `PrefGetPreference`. For details, see “Section 43: Preferences” in the *Palm OS Programmer’s API Reference*. Note that the currently defined sound volume range is 0-sndMaxAmp (64).

However, Tapwave developers are encouraged to use the provided routines for getting and setting the primary volume, and to allow the user to easily change the volume from within any application. If appropriate, developers may want to provide advanced “mixing” controls to independently control the volume of different application sounds, but keep in mind that these are always relative to the user’s primary volume setting.

The only exception to the user’s primary volume are alarm sounds. When an alarm plays, all other sounds are suspended, the speakers are turned on (even if headphones are inserted), and the user’s primary volume setting may be temporarily increased. The alarm state is automatically invoked when the Attention Manager is active. However, applications that do not use the Attention Manager you may need to use the provided APIs to provide the correct behavior for alarms.

Muting is done outside of the main volume control, so when a user unmutes the volume, it returns to the original setting. Tapwave devices offer an additional user interface and API around muting and unmuting, however most applications should not require coding of the mute state.

TwSndGetVolume

Purpose	Gets the current primary volume.
Prototype	`UInt16 TwSndGetVolume(void)`
Result	`Number between 0 and sndMaxAmp (64) indicating the current primary volume.`

Comments	This volume level does not match any Palm OS volume setting or preference. The Palm OS volumes are always relative to this primary setting, with "maximum" volume in Palm OS being the current primary volume.
Header	`TwSound.h`

TwSndSetVolume

Purpose	Sets the primary volume.	
Prototype	`Err TwSndSetVolume(UInt16 newVolume)`	
Parameters	`newVolume`	A number between 0 and `sndMaxAmp` (64) for the new volume. 0 is effectively muted, but differs from the mute state.
Result	`sysErrParamErr if the newVolume is out of range.`	
Comments	Changing this volume level does not change any Palm OS volume setting or preference. The Palm OS volumes are always relative to this primary setting, with "maximum" volume in Palm OS being the current primary volume. (Thus system sounds (clicks, beeps, etc) are set relative to this level: turning down the primary volume also turns down the beeps and clicks.)	
Header	`TwSound.h`	

twNotifySoundVolumeChangedEvent

Purpose	Broadcasts when the volume is adjusted.

Prototype

```
#define twNotifySoundVolumeChangedEvent  'Twsv'

typedef UInt32 TwNotifySoundVolumeChangedDetailsType;

#define twSndVolumeChangedHeadphoneInserted
      0x00010000UL
#define twSndVolumeChangedHeadphoneRemoved
      0x00020000UL
#define twSndVolumeChangedSetVolume
      0x00030000UL
#define twSndVolumeChangedReasonMask
      0x00FF0000UL
```

Parameters	`details`	The notify details pointer is really a 32-bit masked value. The lower 16 bits contain the new sound volume and the upper 16 bits contain the reason the volume was changed.
Comments	This NotifyManager event is sent when `TwSndSetVolume` is called to change the primary volume, or when the volume is changed as a result of inserting or removing the headphones. (Note that the decision to change the volume with headphone insertion may vary depending on yet to be determined values.)	
Header	`TwSound.h`	

Sample

```
UInt32 details = (UInt32) notifyDetailsP;
if ((details & twSndVolumeChangedReasonMask) ==
      twSndVolumeChangedHeadphoneInserted) { }
```

TwSndGetBassBoost

Purpose	Gets the current headphone bass boost level.
Prototype	`UInt16 TwSndGetBassBoost(void)`
Result	`Number between 0 and sndMaxAmp (64) indicating the current bass boost level.`

Comments	This call returns the current bass boost setting. Bass boost applies only to sounds played through the headphones.
Header	`TwSound.h`

TwSndSetBassBoost

Purpose	Sets the current headphone bass boost level.	
Prototype	`Err TwSndSetBassBoost(UInt16 boostLevel)`	
Parameters	`boostLevel`	Either 0 or `smdMaxAmp` (64) for the boost level. 0 means no bass boost, 64 means turn on bass boost.
Result	`sysErrParamErr if the boostLevel is out of range.`	
Header	`TwSound.h`	

TwSndSetMute

Purpose	Mute or unmute the device.	
Prototype	`void TwSndSetMute(Boolean mute, Uint32 unmuteAt)`	
Parameters	`mute`	The new setting, true to mute, false to unmute.
	`unmuteAt`	The time when the mute is canceled. Use zero to mute indefinitely.
Comments	This does not change the result or the behavior of `TwSndGetVolume` and `TwSndSetVolume`. If the primary volume is changed while muted, the changed value will be effective on unmute.	

Header	`TwSound.h`

TwSndGetMute

Purpose	Query the mute setting.	
Prototype	`Boolean TwSndSetMute(UInt32* unmuteAtP)`	
Parameters	`unmuteAtP`	A pointer to a UInt32 that is defined as the time when the mute is scheduled to be canceled. The value will be zero if sound is not muted or if unmute is not scheduled. Pass NULL if you don’t care about this setting.
Result	The mute setting – true if muted, false if not.	
Header	`TwSound.h`	

twNotifyMuteEvent

Purpose	Broadcasts when the device is being muted or unmuted.	
Prototype		

```
#define twNotifyMuteEvent                'Twsm'

typedef struct TwNotifyMuteDetailsTag {
      Int32 muted;
      UInt32 unmuteAt;
} TwNotifyMuteDetailsType;
```

Parameters	`muted`	True if the device is being muted, false if it is being unmuted.
Comments	This NotifyManager event is sent when `TwSndSetMute` is called to change the mute state, or when the mute timer expires and the device unmutes. Applications that provide UI which is synchronized with the system-wide mute state will normally request this notification while they	

	are running.
Header	`TwSound.h`

TwSndPlaySystemSound

Purpose	Play a standard system sound or a Tapwave special sound.	
Prototype	`Err TwSndPlaySystemSound(enum TwSysBeepTag beepID)`	
Parameters	`beepID`	The ID of the sound to play. An invalid ID returns a `sysErrParamErr`.

Comments

Tapwave special sounds are defined by the enum and correspond to new sounds used for feedback during navigation and at other times. You are welcome to use these special sounds in your own applications, but note that the actual sound played may change in future devices. Below is a list of Tapwave's additional beep tags and their interface "meaning."

`twSndBumpedEdge` = hit the edge of a scrollable area

`twSndFollowedLink` = followed a link to another screen or page

`twSndCardInserted` = SD card inserted

`twSndCardRemoved` = SD card removed

`twSndDocked` = HotSync cable plugged in or device inserted in cradle

`twSndUndocked` = HotSync cable unplugged or device removed from cradle

`twSndNextPage` = flipped or scrolled to the next page

`twSndPrevPage` = flipped or scrolled to a previous page

`twSndSyncBegin` = HotSync started

`twSndSyncEnd` = HotSync finished

`twSndEnter` = entered a new folder

`twSndLaunch` = launched an application

`twSndSelection` = selected an item

`twSndLeave` = left a folder

`twSndGraffitiOpen` = Pen input area opened

`twSndGraffitiClose` = Pen input area closed

`twSndRotate` = screen rotated

`twSndBluetoothOn` = Bluetooth enabled

`twSndBluetoothOff` = Bluetooth disabled

`twSndVolumeChange` = sample sound played to demonstrate new volume

`twSndConnect` = Bluetooth connection made

	`twSndGoDoPlay` = animation step sound

Header	`TwSound.h`

TwSndSetAlarmPlaying

Purpose	Invoke (or complete) *alarm mode*. Speakers are turned on even if headphones are inserted. If headphones are not inserted, the primary volume is set to the maximum level. Mute state is not changed.	

Prototype	`Err TwSndSetAlarmPlaying(Boolean isAlarm)`
	`    TAL_TRAP(trapTwSndSetAlarmPlaying);`	

Parameters	`isAlarm`	True to turn on alarm mode, false to turn alarm mode off again.
Comments	Applications which use the Attention Manager do not need to call this function, the Attention Manager sets the proper mode before asking the application to play its sound. However, many apps provide UI that demonstrates the alarm sound when the user chooses an alarm – these apps should call this function before and after playing the demonstration sound. Also, applications that play alarms that do not use Attention Manager may need to call this function to get proper alarm behavior.	
Header	`TwSound.h`	

2.Sound Device API

In addition to the TwSnd API additions, a Tapwave device API exists for feeding data directly to the audio mixer (the mixer is responsible for combining sound data from multiple sources into a single stream which is heard through the speakers or headphones).

The “mixer” device (whose specific device API is described in TwVdMixer.h) can be opened using TwDeviceOpen, configured using TwDeviceSetProperty, written to using TwDeviceWrite, and closed using TwDeviceClose. When using TwDeviceWrite the raw sample data is written to the buffer associated with the device; the audio mixer uses this buffer to resample and mix into the audio output buffer.

Unlike the SndStream API, there is no notion of “starting” or “stopping” the stream. The SndStream API is a “pull” model API – a callback is invoked by a system thread to “pull” data into a buffer at a periodic rate. The mixer device API is a “push” model API – the application can invoke TwDeviceWrite at whatever rate it desires (note that TwDeviceWrite will block until all of the data given it is written into the buffer). Please note that the mixer always mixes at a specific sample rate with a specific number of samples per buffer. If your buffer doesn’t have enough data present then the mixer will ignore your buffer until the next sample period. Use the TW_VD_MIXER_BUFFER_SAMPLES to determine how many samples must be written to the device buffer to satisfy the mixer.

Here is a list of the properties supported by TwDeviceSetProperty and TwDeviceGetProperty for the mixer device:

Property	**Description**
TW_VD_MIXER_CONFIG	Set/Get the configuration for this mixer stream. The type of the argument data must be this:

```
typedef struct TwVdMixerConfigProperty {

  UInt32 sampleRate;  // e.g. 44100

  UInt32 format;      // See SoundMgr.h SndFormatType

  UInt32 type;        // See SoundMgr.h SndSampleType

  UInt32 width;       // See SoundMgr.h SndStreamWidth

} TwVdMixerConfigProperty;
```

TW_VD_MIXER_VOLUME	Set/Get the volume for the mixer. The argument data must be a UInt32. The volume range is the same as the SndStream volume range.
TW_VD_MIXER_PAN	Set/Get the mixer pan position. The argument data must be a UInt32. The pan position is the same as the SndStream pan position.
TW_VD_MIXER_BUFFER_BYTES	Get the mixer buffer size. This will return the number of bytes of buffering used by the audio mixer. The argument data must be a UInt32. Note that this value will be a constant for a given hardware configuration.

TW_VD_MIXER_BUFFER_FRAMES	Get the mixer buffer size, but in samples per buffer instead of bytes per buffer. The argument data must be a UInt32. This value will depend on the actual configuration of the stream. This Get will return an error if the stream has not been configured.