Technical Note TN1198

SndPlayDoubleBuffer and Carbon

CONTENTS

This Technote describes the removal of the SndPlayDoubleBuffer and other APIs from the Carbon API set and what can be done to make existing sound code based on these now-defunct APIs Carbon compatible.

This Note is directed at application developers who have code that calls functions that are not in the Carbon Sound Manager and who want to move that code over to Carbon.

[June 12 2002]

The loss of the APIs

These sound functions are not in Carbon:

- SndControl
- SndStartFilePlay
- SndPauseFilePlay
- SndStopFilePlay
- SndPlayDoubleBuffer
- MACEVersion
- Comp3to1
- Exp1to3
- Comp6to1
- Exp1to6
- AudioGetVolume
- AudioSetVolume
- AudioGetMute
- AudioSetMute
- AudioSetToDefaults
- AudioGetInfo
- AudioGetBass
- AudioSetBass
- AudioGetTreble
- AudioSetTreble
- AudioGetOutputDevice
- AudioMuteOnEvent
- SndRecordToFile
- SPBRecordToFile

These SndCommand numbers are not supported by the Carbon Sound Manager:

- initCmd
- freeCmd
- totalLoadCmd
- loadCmd
- freqDurationCmd
- restCmd
- freqCmd
- ampCmd
- timbreCmd
- getAmpCmd
- waveTableCmd
- phaseCmd
- rateCmd
- continueCmd
- doubleBufferCmd
- getRateCmd
- sizeCmd /*obsolete command*/
- convertCmd /*obsolete MACE command*/

If you have code that relies on any of the above calls or SndCommands, you will have to rewrite it if you want to make your application Carbon compatible. This Note will describe how to duplicate the functionality of these APIs with Carbon-compatible code.

Back to top

Why they were removed

The low-level calls were removed because there are better, more compatible, ways of accomplishing what these calls accomplished. In some cases, the calls had ceased to be useful so their removal should not affect any modern code base.

The high-level calls, `SndStartFilePlay`, `SPBRecordToFile`, etc., were removed because their functionality is largely subsumed by QuickTime. Using QuickTime in place of these routines should increase the functionality of your program (for instance, you will be able to play a more varied list of sound files) without adding much work or code to your existing code base.

The `SndCommands` were removed because they either operate on non-wave data which is no long supported, or there are newer `SndCommands` that supercede their functionality.

Back to top

What they did

This is a brief description of what these functions did. For a complete description, see Inside Macintosh: Sound.

- `SndControl`
  - In Sound Manager 2.0 this returned information about the sound hardware.
- `SndStartFilePlay`
  - Starts a file playing from disk. This function is basically a wrapper around `SndPlayDoubleBuffer`.
- `SndPauseFilePlay`
  - Toggles the playing state of a file that is being played by `SndStartFilePlay`.
- `SndStopFilePlay`
  - Stops a sound that is being played by `SndStartFilePlay`.
- `SndPlayDoubleBuffer`
  - Plays a sound by first playing one buffer of audio and then a second, allowing the alternate buffer to be filled in from disk (or wherever) while the other buffer plays. Allows large sound files to be played with only a minimum use of RAM.
- `MACEVersion`
  - Gets the version of the MACE compressor/decompressor.
- `Comp3to1`
  - Compresses a sound with MACE 3:1.
- `Exp1to3`
  - Decompresses a sound compressed with MACE 3:1.
- `Comp6to1`
  - Compresses a sound with MACE 6:1.
- `Exp1to6`
  - Decompresses a sound compressed with MACE 6:1.
- `AudioGetVolume`
  - Called by the Sound Manager to get an output component's volume.
- `AudioSetVolume`
  - Called by the Sound Manager to set an output component's volume.
- `AudioGetMute`
  - Called by the Sound Manager to get the mute state of an output component.
- `AudioSetMute`
  - Called by the Sound Manager to set the mute state of an output component.
- `AudioSetToDefaults`
  - Called by the Sound Manager to return an output component to its defaults.
- `AudioGetInfo`
  - Called by the Sound Manager to get information about an output component.
- `AudioGetBass`
  - Called by the Sound Manager to get an output component's bass volume.
- `AudioSetBass`
  - Called by the Sound Manager to set an output component's bass volume.
- `AudioGetTreble`
  - Called by the Sound Manager to get an output component's treble volume.
- `AudioSetTreble`
  - Called by the Sound Manager to set an output component's treble volume.
- `AudioGetOutputDevice`
  - Not documented.
- `AudioMuteOnEvent`
  - Not documented.
- `SndRecordToFile`
  - Records sound data to a file in a synchronous operation.
- `SPBRecordToFile`
  - Records sound data to a file, optionally doing it asynchronously.

- `initCmd`
  - Not documented.
- `freeCmd`
  - Not documented.
- `totalLoadCmd`
  - Sent using the obsolete `SndControl` function, it reported the total CPU load factor for all existing sound activity and for a new sound channel having the initialization parameters specified in `param2`.
- `loadCmd`
  - Sent using the obsolete `SndControl` function, it reported the percentage of CPU processing power that the sound channel specified in `param2` would require.
- `freqDurationCmd`
  - Play the note specified in `param2` for the duration specified in `param1`.
- `freqCmd`
  - Change the frequency (or pitch) of a sound. If no sound is currently playing, then `freqCmd` causes the Sound Manager to begin playing indefinitely at the frequency specified in `param2`. Could be used to loop a sampled-sound data sound installed with a `soundCmd`.
- `restCmd`
  - Rest a channel for a specified duration.

- `ampCmd`
  - Change the amplitude (or loudness) of a sound.
- `timbreCmd`
  - Change the timbre (or tone) of a sound currently being defined using square-wave data.
- `getAmpCmd`
  - Determine the current amplitude (or loudness) of a sound.
- `waveTableCmd`
  - Install a wave table as a voice in the specified channel.
- `phaseCmd`
  - Not documented.
- `rateCmd`
  - Set the rate of a sampled sound that is currently playing, thus effectively altering its pitch and duration. Your application can set a rate of 0 to pause a sampled sound that is playing. The new rate is set to the value specified in `param2`, which is interpreted relative to 22 kHz.
- `continueCmd`
  - Not documented.
- `doubleBufferCmd`
  - This is the command used by `SndPlayDoubleBuffer`.
- `getRateCmd`
  - Determine the sample rate of the sampled sound currently playing. The current rate of the channel is returned in a `Fixed` variable whose address you pass in `param2` of the sound command. The values returned are always relative to the 22 kHz sampling rate, as with the `rateCmd` sound command.
- `sizeCmd /*obsolete command*/`
  - Not documented.
- `convertCmd /*obsolete MACE command*/`
  - Not documented.

Back to top

How they can be replaced

- `SndControl`
  - This call was made obsolete by Sound Manager 3.0 and should be replaced by calls to `Gestalt`.
- `SndStartFilePlay`
  - Can be mimicked with QuickTime.
- `SndPauseFilePlay`
  - When using QuickTime, use QuickTime's control commands.
- `SndStopFilePlay`
  - When using QuickTime, use QuickTime's control commands.
- `SndPlayDoubleBuffer`
  - Can be mimicked with `bufferCmd` and `callBackCmd`, or to some extent, QuickTime.
- `MACEVersion`
  - This call should never be needed. The current version of MACE is 1.0.2 and it hasn't changed in years.
- `Comp3to1`
  - Can be replaced with the appropriate calls to the `SoundConverter` APIs.
- `Exp1to3`
  - Can be replaced with the appropriate calls to the `SoundConverter` APIs.
- `Comp6to1`
  - Can be replaced with the appropriate calls to the `SoundConverter` APIs.
- `Exp1to6`
  - Can be replaced with the appropriate calls to the `SoundConverter` APIs.
- `AudioGetVolume`
  - Can be replaced with `SoundComponentGetInfo` and the `siHardwareVolume` selector.
- `AudioSetVolume`
  - Can be replaced with `SoundComponentSetInfo` and the `siHardwareVolume` selector.
- `AudioGetMute`
  - Can be replaced with `SoundComponentGetInfo` and the `siHardwareMute` selector.
- `AudioSetMute`
  - Can be replaced with `SoundComponentSetInfo` and the `siHardwareMute` selector.
- `AudioSetToDefaults`
  - No replacement.
- `AudioGetInfo`
  - Can be replaced with calls to `SoundComponentGetInfo` and the appropriate selectors.
- `AudioGetBass`
  - Can be replaced with `SoundComponentGetInfo` and the `siHardwareBass` selector.
- `AudioSetBass`
  - Can be replaced with `SoundComponentSetInfo` and the `siHardwareBass` selector.
- `AudioGetTreble`
  - Can be replaced with `SoundComponentGetInfo` and the `siHardwareTreble` selector.
- `AudioSetTreble`
  - Can be replaced with `SoundComponentSetInfo` and the `siHardwareTreble` selector.
- `AudioGetOutputDevice`
  - No replacement.
- `AudioMuteOnEvent`
  - No replacement.
- `SndRecordToFile`
  - Can be mimicked with Sequence Grabber (QuickTime).
- `SPBRecordToFile`
  - Can be mimicked with appropriate calls `SPBRecord` and `PBWriteAsync` for the asynchronous case. QuickTime can also be used to replace it.

- `initCmd`
  - No replacement.
- `freeCmd`
  - No replacement.
- `totalLoadCmd`
  - No replacement, but it was not accurate or useful starting with Sound Manager 3.1.
- `loadCmd`
  - No replacement, but it was not accurate or useful starting with Sound Manager 3.1.

- freqDurationCmd
  - Replace the pitch shift functionality with rateMultiplierCmd.
- freqCmd
  - Replace the pitch shift and duration functionality with rateMultiplierCmd. You can no longer loop a sound. To loop a sound play it over and over again using bufferCmd and callBackCmd.
- restCmd
  - No replacement as square wave data sound and wave table data sound are not supported by the Carbon Sound Manager.
- ampCmd
  - Use volumeCmd instead.
- timbreCmd
  - No replacement as square wave data sound and wave table data sound are not supported by the Carbon Sound Manager.
- getAmpCmd
  -
- waveTableCmd
  - No replacement as wave table data sound are not supported by the Carbon Sound Manager.
- phaseCmd
  - No replacement.
- rateCmd
  - Use rateMultiplierCmd instead.
- continueCmd
  - No replacement.
- doubleBufferCmd
  - This is the command used by SndPlayDoubleBuffer. No direct replacement. You can use bufferCmd and callBackCmd to simulate, see Replacing SndPlayDoubleBuffer below.
- getRateCmd
  - Use getRateMultiplierCmd instead.
- sizeCmd /*obsolete command*/
  - No replacement.
- convertCmd /*obsolete MACE command*/
  - No replacement.

Back to top

Replacing SndPlayDoubleBuffer

Replacing SndPlayDoubleBuffer is a bit tricky because of the subtle ways it does its work. Replacing it a simple stream of bufferCmds and callBackCmds does not do exactly the same thing for code that was expecting the real SndPlayDoubleBuffer.

This is because the real SndPlayDoubleBuffer call is a single command in the sound channel's queue, a doubleBufferCmd, but the replacement is two commands. Furthermore, the doubleBufferCmd stays in the channel's queue until the sound is done playing, but the bufferCmd and callBackCmd are constantly being added and removed. This is an issue for any commands that might be waiting in the queue after the doubleBufferCmd and the code that relies on those commands being after the doubleBufferCmd. This doesn't immediately kill the prospect of simulating SndPlayDoubleBuffer, but it means that the work is a little more complicated for the simulating code.

The first issue is that information about the currently playing sound will need to be kept for each sound channel. This information is kept by the Sound Manager when using the real SndPlayDoubleBuffer.

```
// Structs
struct PerChanInfo {
    QElemPtr            qLink;    /* next queue entry */
    short               qType;    /* queue type = 0 */
    short               stopping;
    #if DEBUG
        OSType          magic;
    #endif
    SndCallBackUPP       usersCallBack;
    SndDoubleBufferHeader   theParams;
    CmpSoundHeader       soundHeader;
};
typedef struct PerChanInfo   PerChanInfo;
typedef struct PerChanInfo * PerChanInfoPtr;

// Globals
    Boolean             gNMRecBusy;
    NMRecPtr            gNMRecPtr;
    QHdrPtr             gFreeList;
    Ptr                 gSilenceTwos;
    Ptr                 gSilenceOnes;
```

The queue structure is used to keep track of the per-channel sound information, such as the format of the sound, the parameters to SndPlayDoubleBuffer, the callback function that was originally in the sound channel (before the simulating code users it for its own use) and some housekeeping information. This queue structure will allow us to enqueue the channel information using PBEnqueue at interrupt time so that at task time we can dispose of the per-channel structure and associated memory.

The simulation has to set up its simple state machine and the first buffer of sound has to be played. That code looks like this:

```
// This function is only callable at system task time.
//    Note: CarbonSndPlayDoubleBuffer calls NewPtrClear,
//          which is only callable at system task time,
```

```
//          this means that CarbonSndPlayDoubleBuffer itself
//          is only callable at system task time.
OSErr    CarbonSndPlayDoubleBuffer (SndChannelPtr chan,
  SndDoubleBufferHeaderPtr theParams) {
    OSErr                           err;
    CompressionInfo                   compInfo;
    PerChanInfoPtr                   perChanInfoPtr;
    SndCommand                       playCmd;
    SndCommand                       callBack;

    if (nil == chan) {
        err = badChannel;
        goto exit;
    }

    if (nil == theParams) {
        err = paramErr;
        goto exit;
    }

    if (nil == gFreeList) {
        // This can't ever be disposed since we don't know when
        // we might need to use it (at interrupt time)
        gFreeList = (QHdrPtr)NewPtrClear (sizeof (QHdr));
        err = MemError ();
        if (noErr != err) goto exit;
    }

    if (nil == gSilenceOnes) {
        short        i;
        // This can't ever be disposed since we don't know when
        // we might need to use it (at interrupt time)
        gSilenceOnes = NewPtr (kBufSize);
        err = MemError ();
        if (noErr != err) goto exit;
        for (i = 0; i < kBufSize; i++) {
            gSilenceOnes[i] = (char)0x80;
        }
    }

    if (nil == gSilenceTwos) {
        // This can't ever be disposed since we don't know when
        // we might need to use it (at interrupt time)
        gSilenceTwos = NewPtrClear (kBufSize);
        err = MemError ();
        if (noErr != err) goto exit;
    }

    if (nil == gNMRecPtr) {
        // This can't ever be disposed since we don't know when
        // we might need to use it (at interrupt time)
        gNMRecPtr = (NMRecPtr)NewPtr (sizeof (NMRec));
        err = MemError ();
        if (noErr != err) goto exit;

        // Set up our NMProc info that will dispose of most
        // (but not all) of our memory
        gNMRecPtr->qLink = nil;
        gNMRecPtr->qType = 8;
        gNMRecPtr->nmFlags = 0;
        gNMRecPtr->nmPrivate = 0;
        gNMRecPtr->nmReserved = 0;
        gNMRecPtr->nmMark = nil;
        gNMRecPtr->nmIcon = nil;
        gNMRecPtr->nmSound = nil;
        gNMRecPtr->nmStr = nil;
        gNMRecPtr->nmResp = NewNMProc (NMResponseProc);
        gNMRecPtr->nmRefCon = 0;
    }

    perChanInfoPtr = (PerChanInfoPtr)NewPtr (sizeof (PerChanInfo));
    err = MemError ();
    if (noErr != err) goto exit;

    // Init basic per channel information
    perChanInfoPtr->qLink = nil;
    perChanInfoPtr->qType = 0;                // not used
    perChanInfoPtr->stopping = 0;
    #if DEBUG
        perChanInfoPtr->magic = 'SANE';
    #endif

    perChanInfoPtr->theParams = *theParams;
    // Have to remember the user's callback function from the sound because
    // we are going to overwrite it with our own callback function.
    perChanInfoPtr->usersCallBack = chan->callBack;

    // Set up the sound header for the bufferCmd that will be used to play
    // the buffers passed in by the SndPlayDoubleBuffer call.
```

```
    perChanInfoPtr->soundHeader.samplePtr =
         (Ptr)(theParams->dbhBufferPtr[0]->dbSoundData);
    perChanInfoPtr->soundHeader.numChannels =
         theParams->dbhNumChannels;
    perChanInfoPtr->soundHeader.sampleRate =
         theParams->dbhSampleRate;
    perChanInfoPtr->soundHeader.loopStart = 0;
    perChanInfoPtr->soundHeader.loopEnd = 0;
    perChanInfoPtr->soundHeader.encode = cmpSH;
    perChanInfoPtr->soundHeader.baseFrequency = kMiddleC;
    perChanInfoPtr->soundHeader.numFrames =
         (unsigned long)theParams->dbhBufferPtr[0]->dbNumFrames;
    //    perChanInfoPtr->soundHeader.AIFFSampleRate = 0;          // unused
    perChanInfoPtr->soundHeader.markerChunk = nil;
    perChanInfoPtr->soundHeader.futureUse2 = nil;
    perChanInfoPtr->soundHeader.stateVars = nil;
    perChanInfoPtr->soundHeader.leftOverSamples = nil;
    perChanInfoPtr->soundHeader.compressionID =
         theParams->dbhCompressionID;
    perChanInfoPtr->soundHeader.packetSize =
         (unsigned short)theParams->dbhPacketSize;
    perChanInfoPtr->soundHeader.snthID = 0;
    perChanInfoPtr->soundHeader.sampleSize =
         (unsigned short)theParams->dbhSampleSize;
    perChanInfoPtr->soundHeader.sampleArea[0] = 0;

    // Is the sound compressed?  If so, we need to treat
    // theParams as a SndDoubleBufferHeader2Ptr.
    if (0 != theParams->dbhCompressionID) {
        // Sound is compressed
        err = GetCompressionInfo (theParams->dbhCompressionID,
               ((SndDoubleBufferHeader2Ptr)theParams)->dbhFormat,
               theParams->dbhNumChannels,
               theParams->dbhSampleSize,
               &compInfo);
        if (noErr != err) goto exitDispose;

        perChanInfoPtr->soundHeader.format = compInfo.format;
    } else {
        // Sound is not compressed
        perChanInfoPtr->soundHeader.format = kSoundNotCompressed;
    }

    playCmd.cmd = bufferCmd;
    playCmd.param1 = 0;          // unused
    playCmd.param2 = (long)&perChanInfoPtr->soundHeader;

    callBack.cmd = callBackCmd;
    callBack.param1 = 0;          // which buffer to fill, 0 buffer, 1, 0, ...
    callBack.param2 = (long)perChanInfoPtr;

    // Install our callback function pointer straight into
    // the sound channel structure
    if (nil == gCarbonSndPlayDoubleBufferCallBackUPP) {
        gCarbonSndPlayDoubleBufferCallBackUPP =
             NewSndCallBackProc (CarbonSndPlayDoubleBufferCallBackProc);
    }

    chan->callBack = gCarbonSndPlayDoubleBufferCallBackUPP;

    if (nil == gCarbonSndPlayDoubleBufferCleanUpUPP) {
        #if !TARGET_API_MAC_CARBON
            gCarbonSndPlayDoubleBufferCleanUpUPP =
                 NewSndCallBackProc (CarbonSndPlayDoubleBufferCleanUpProc);
        #endif
    }

    err = SndDoCommand (chan, &playCmd, true);
    if (noErr != err) goto exitDispose;

    err = SndDoCommand (chan, &callBack, true);
    if (noErr != err) goto exitDispose;

exit:
    return err;

exitDispose:
    if (nil != perChanInfoPtr)
        DisposePtr ((Ptr)perChanInfoPtr);
    goto exit;
}
```

In Carbon there is no UPP for the `SndDoubleBackProc`, but that's OK. Since all code in Carbon is PowerPC, and this code will be compiled into the calling program (therefore not needing to worry about CFM<->Mach-O calling conventions) the `SndDoubleBackProc` will just be treated as a regular C function pointer.

The callback function that tells the user's code to refill the now empty buffer and begins playing the alternate buffer looks

like this:

```
static pascal void    CarbonSndPlayDoubleBufferCallBackProc
              (SndChannelPtr theChannel, SndCommand * theCallBackCmd) {
    SndDoubleBufferHeaderPtr        theParams;
    SndDoubleBufferPtr                emptyBuf;
    SndDoubleBufferPtr                nextBuf;
    PerChanInfoPtr                    perChanInfoPtr;
    SndCommand                        playCmd;

    perChanInfoPtr = (PerChanInfoPtr)(theCallBackCmd->param2);
    #if DEBUG
        if (perChanInfoPtr->magic != 'SANE')
                    DebugStr("\pBAD in CarbonSndPlayDoubleBufferCallBackProc");
    #endif
    if (true == perChanInfoPtr->stopping) goto exit;

    theParams = &(perChanInfoPtr->theParams);

    // The buffer that just played and needs to be filled
    emptyBuf = theParams->dbhBufferPtr[theCallBackCmd->param1];

    // Clear the ready flag
    emptyBuf->dbFlags ^= dbBufferReady;

    // This is the buffer to play now
    nextBuf = theParams->dbhBufferPtr[!theCallBackCmd->param1];

    // Check to see if it is ready, or if we have to wait a bit
     if (nextBuf->dbFlags & dbBufferReady) {
  perChanInfoPtr->soundHeader.numFrames =
     (unsigned long)nextBuf->dbNumFrames;
          perChanInfoPtr->soundHeader.samplePtr = Ptr)(nextBuf->dbSoundData);

// Flip the bit telling us which buffer is next
theCallBackCmd->param1 = !theCallBackCmd->param1;

// If this isn't the last buffer, call the user's fill routine
     if (!(nextBuf->dbFlags & dbLastBuffer)) {
          #if TARGET_API_MAC_CARBON
          // Declare a function pointer to the user's double back proc
          void (*doubleBackProc)(SndChannel*, SndDoubleBuffer*);

               // Call user's double back proc
              doubleBackProc = (void*)theParams->dbhDoubleBack;
              (*doubleBackProc) (theChannel, emptyBuf);
            #else
              CallSndDoubleBackProc (theParams->dbhDoubleBack, theChannel, emptyBuf);
            #endif
        } else {
            // Call our clean up proc when the last buffer finishes
            theChannel->callBack = gCarbonSndPlayDoubleBufferCleanUpUPP;
        }
    } else {
        // We have to wait for the buffer to become ready.
        // The real SndPlayDoubleBuffer would play a short bit of silence
        // waiting for the user to read the audio from disk,
        // so that's what we do here.
        #if DEBUG
            DebugStr ("\p buffer is not ready!");
        #endif
        // Play a short section of silence so that we can check the
        // ready flag again
        if (theParams->dbhSampleSize == 8) {
            perChanInfoPtr->soundHeader.numFrames =
                   (UInt32)(kBufSize / theParams->dbhNumChannels);
            perChanInfoPtr->soundHeader.samplePtr = gSilenceOnes;
        } else {
            perChanInfoPtr->soundHeader.numFrames =
                   (UInt32)(kBufSize / (theParams->dbhNumChannels *
                   (theParams->dbhSampleSize / 8)));
            perChanInfoPtr->soundHeader.samplePtr = gSilenceTwos;
        }
    }

    // Insert our callback command
    InsertSndDoCommand (theChannel, theCallBackCmd);

    // Play the next buffer
    playCmd.cmd = bufferCmd;
    playCmd.param1 = 0;
    playCmd.param2 = (long)&(perChanInfoPtr->soundHeader);
    InsertSndDoCommand (theChannel, &playCmd);

exit:
    return;
}
```

There is a further callback function that runs only once the application has signaled that we have played the last buffer of its data. This function queues the per-channel sound information so that the Notification Manager callback can dispose of the per-channel memory.

```
static pascal void    CarbonSndPlayDoubleBufferCleanUpProc
             (SndChannelPtr theChannel, SndCommand * theCallBackCmd) {
    PerChanInfoPtr perChanInfoPtr;

    perChanInfoPtr = (PerChanInfoPtr)(theCallBackCmd->param2);
    #if DEBUG
        if (perChanInfoPtr->magic != 'SANE') DebugStr("\pBAD in
            CarbonSndPlayDoubleBufferCleanUpProc");
    #endif

    // Put our per channel data on the free queue so we can
    // clean up later
    Enqueue ((QElemPtr)perChanInfoPtr, gFreeList);

    // Have to put the user's callback proc back so
    // they get called when the next buffer finishes
    theChannel->callBack = perChanInfoPtr->usersCallBack;

    // Have to install our Notification Manager routine so that
    // we can clean up the gFreeList
    if (!OTAtomicSetBit (&gNMRecBusy, 0)) {
        NMInstall (gNMRecPtr);
    }
}
```

The next issue is that because the simulating code will not be called once the sound has finished, it doesn't have a good opportunity to clean up after itself because it will be called at interrupt time when it wants to dispose of the associated memory for the completed sound. This can be partially alleviated by using the Notification Manager to dispose of most of the memory, but there is no easy way to clean up the Notification Manager record that is allocated for this.

The Notification Manager callback code looks like this:

```
static pascal void NMResponseProc (NMRecPtr nmReqPtr) {
    PerChanInfoPtr                  perChanInfoPtr;
    OSErr                          err;

    NMRemove (nmReqPtr);
    gNMRecBusy = false;

    do {
        perChanInfoPtr = (PerChanInfoPtr)gFreeList->qHead;
        if (nil != perChanInfoPtr) {
            err = Dequeue ((QElemPtr)perChanInfoPtr, gFreeList);
            if (noErr == err) {
                DisposePtr ((Ptr)perChanInfoPtr);
            }
        }
    } while (nil != perChanInfoPtr && noErr == err);
}
```

Because existing code will assume that once it has called `SndPlayDoubleBuffer` that it can install a `callBackCmd` without affecting the playing of the sound, the code replacing `SndPlayDoubleBuffer` must insert its `bufferCmd`s and `callBackCmd`s at the head of the sound queue. This means directly manipulating the sound channel's command queue.

The code to do that looks like:

```
static void    InsertSndDoCommand (SndChannelPtr chan, SndCommand * newCmd) {
    if (-1 == chan->qHead) {
        chan->qHead = chan->qTail;
    }

    if (1 <= chan->qHead) {
        chan->qHead--;
    } else {
        chan->qHead = chan->qLength - 1;
    }

    chan->queue[chan->qHead] = *newCmd;
}
```

This also means that `SndDoImmediate` must be wrapped so as to allow the original code to use the `quietCmd` as it always has. Here is code that shows how to wrap `SndDoImmediate` to make sure that a `quietCmd` stops the sound and

calls the correct function for any `callBackCmd` that might have been installed after the call to `SndPlayDoubleBuffer`.

```
// Remember this routine could be called at interrupt time,
// so don't allocate or deallocate memory.
OSErr    MySndDoImmediate (SndChannelPtr chan, SndCommand * cmd) {
    PerChanInfoPtr                  perChanInfoPtr;

    // Is this being called on one of the sound channels we are manipulating?
    // If so, we need to pull our callback out of the way so
    // that the user's commands run
    if (nil != gFreeList && gCarbonSndPlayDoubleBufferCallBackUPP ==
                        chan->callBack) {
        if (quietCmd == cmd->cmd || flushCmd == cmd->cmd) {
            // We know that our callBackCmd is the first item in the queue
            // if this is our channel
            perChanInfoPtr = (PerChanInfoPtr)
                    (chan->queue[chan->qHead].param2);
            #if DEBUG
                if (perChanInfoPtr->magic != 'SANE')
                        DebugStr("\pBAD in MySndDoImmediate");
            #endif
            perChanInfoPtr->stopping = true;
            Enqueue ((QElemPtr)perChanInfoPtr, gFreeList);
            if (! OTAtomicSetBit (&gNMRecBusy, 0)) {
                NMInstall (gNMRecPtr);
            }
            chan->callBack = perChanInfoPtr->usersCallBack;
        }
    }

    return (SndDoImmediate (chan, cmd));
}
```

Back to top

SndStartFilePlay

If you use `SndStartFilePlay` to play sound resources (resources of type `'snd '`) using limited amounts of memory (rather than loading the entire sound into memory and playing it), your only solution is to write a wrapper around the `CarbonSndPlayDoubleBuffer` code using `ReadPartialResource` to extract only a portion of the sound resource at a time.

QuickTime doesn't give you the option of playing a resource handle whose data hasn't been completely loaded. QuickTime will play a sound resource, but it expects that the resource has been fully loaded into memory.

If you use `SndStartFilePlay` to play sound files from disk, this is probably a good fit for QuickTime. You can even have QuickTime start playing the sound from a specific time, just like `SndStartFilePlay`, though you have to poll to know when the sound is done, there is no QuickTime callback for this information.

The code to have QuickTime play a file from disk looks like:

```
    OSErr           err;
    Movie           theSound;
    short           fileRefNum;

    err = OpenMovieFile (&theSpec, &fileRefNum, fsRdPerm);

    if (noErr == err) {
          err = NewMovieFromFile (&theSound, fileRefNum, 0, nil,
             newMovieActive, nil);
    }

    if (noErr == err) {
        GoToBeginningOfMovie (theSound);
    }

    if (noErr == err) {
        StartMovie (theSound);
    }

    if (noErr == err) {
        while (!IsMovieDone (theSound) {
            MoviesTask (theSound, 0);
        }
    }
```

This code will open a file on disk, pointed to by a `FSSpec`, and play it synchronously from the beginning of the sound to the end. `SndStartFilePlay` required you to already have the file open and pass it a file reference number, so you probably already have a `FSSpec` to the file you want to play.

If you want to start the sound at some place other than the start, you would use QuickTime's `SetMovieTime` function to set the current time in the movie.

If you want to play the sound asynchronously, then you will need to make `theSound` a global and in your main event loop call `MoviesTask` about every quarter of a second to keep the movie running with any glitches. What this implies is that if you cannot call `MoviesTask`, for instance, because your user is holding down the mouse button and you are inside a call to `TrackDrag`, then the sound will stop. If that is unacceptable, then you will probably want to convert to using the `CarbonSndPlayDoubleBuffer` code which does not require task time to continue playing a sound.

Back to top

`SndRecordToFile` and `SPBRecordToFile`

If you use `SndRecordToFile` to record sound to disk, the easiest way to convert to Carbon is to use QuickTime. The following code shows how you would use QuickTime's Sequence Grabber to record audio in a synchronous manner.

```
SGChannel               sgSoundChan;
ComponentInstance        sgSoundComp;
short                   numChannels,
                       sampleSize;
OSType                  compressionType,
                       inputSource;

err = SGInitialize (sgSoundComp);

if (err == noErr) {
    err = SGNewChannel (sgSoundComp, SoundMediaType, &sgSoundChan);
}

if (err == noErr) {
    err = SGSetChannelUsage (sgSoundChan, seqGrabRecord);
}

if (err == noErr) {
    err = SGSetSoundInputRate (sgSoundChan, sampleRate);
}

if (err == noErr) {
    err = SGSetSoundInputParameters
              (sgSoundChan, sampleSize, numChannels, compressionType);
}

if (err == noErr) {
    err = SPBSetDeviceInfo
              (SGGetSoundInputDriver (sgSoundChan),
              siOSTypeInputSource, &inputSource);
}

if (err == noErr) {
    err = SGSoundInputDriverChanged (sgSoundChan);
}

if (err == noErr) {
    err = SGSetDataOutput (sgSoundComp, &theSpec, seqGrabToDisk);
}

if (err == noErr) {
    err = SGStartRecord (sgSoundComp);
}

if (err == noErr) {
    EventRecord        event;
    Boolean            done = false;

    while (!done && err == noErr) {
        WaitNextEvent (mDownMask | keyDownMask, &event, 6, nil);
        err = SGIdle (sgSoundComp);
        switch (event.what) {
            case mouseDown:
            case keyDown:
                done = true;
                break;
        }
    }
    err = SGStop (sgSoundComp);
}

if (sgSoundComp != nil) {
    err = CloseComponent (sgSoundComp);
}
```

If you would prefer to use the Sequence Grabber's user interface to have the user configure the recording (which is recommended if you don't have your own interface already) then you can skip the calls to `SGSetSoundInputRate`, `SGSetSoundInputParameters`, and `SPBSetDeviceInfo` with the `siOSTypeInputSource` selector, and

SGSoundInputDriverChanged and replace them all with a single call to SGSettingsDialog.

If you want to record asynchronously (something that SndRecordToFile will not do but SPBRecordToFile would), then you will need to make sgSoundComp a global and call SGIdle from your main event loop at least once every quarter of a second.

One nice feature of using the Sequence Grabber to record is that it will record in any compression format currently installed on the Mac. It does not limit you to only MACE 3:1 and MACE 6:1 compression.

Back to top

Summary

Many of the Sound Manager functions not in Carbon were not used and so their loss will not cause any difficulties. For the other functions, QuickTime and a little extra Sound Manager Carbon compatible code is all that is required.

Since QuickTime is very easy to use (at least in these cases) and has most of the functionality of the non-Carbon Sound Manager calls, converting your code to use QuickTime when converting the rest of your code to Carbon should not be difficult.

For those applications that made heavy use of SndPlayDoubleBuffer and required its interrupt driven nature to deliver uninterrupted audio, the CarbonSndPlayDoubleBuffer and associated functions allow you to do this with a minimum of change to your existing code base.

Back to top

References

Carbon Developer Documentation

Technote 1108: Unknown Sound Features

Technote 1048: Some Sound Advice: Getting the Most Out of the Sound Manager

Inside Macintosh: Sound

Back to top

Downloadables

	Acrobat version of this Note (96K).	Download
	CarbonSndPlayDoubleBuffer code	Download

Back to top
