DSOUND.DLL 3.0 API Programmer's Reference
Aaron Wallace
September, 1991

[Shareware Beta release Notice:

This version is substantially different from what I had planned on releasing for 3.0. First, it uses a sound generation algorithm I've been told does not infringe on RealSound's patent. Second, I'd like to get rid of the whole delay bit, so I'm testing a new pacing method that doesn't use a delay loop. It seems to work within a fudge factor or so... Similarly, I didn't want to have a separate .ASM driver for each output device, so I eliminated the dissimilarities between Standard and Enhanced mode. This should make the migration to 3.1 much easier... Let me know how this driver works, in all modes, for all devices, and on all computers!]

DSOUND.DLL is a Windows 3.0 .DLL that provides a device-independent API for playing digitalized waveform sound, 8 bits per sample, at sampling rates up to 22 KHz. The .DLL supports the PC internal speaker, the Disney Software Sound Source, and the Creative Labs Sound Blaster. Both Standard and Enhanced modes are supported for Windows 3.0 and 3.1 (tested with Beta I). The .DLL may in fact work in Real mode, but such use is now unsupported. The DSound Control Panel (SNDCNTRL.EXE) won't. Currently background playback is not supported (due mainly to hardware and Windows limitations), but I suspect it will be supported in a future release, perhaps involving Multimedia Windows support.

DSOUND.DLL is currently used in several shareware applications (BartEyes II and SoundTool come to mind) and at least one commercial application (Aristosoft's Wired for Sound). The shareware version of the .DLL and the control panel (from now on all references to "the control panel" are to the DSound Control Panel; the Windows control panel buys you little in terms of sound) may be distributed with such shareware as long as it's made clear that the registration for the shareware program does not include registration for DSOUND.DLL. I distribute DSOUND.DLL as "Sounder," which can be registered for $3. Users who have Wired for Sound or other commercial programs based on DSOUND.DLL need not register the shareware version; in fact, the .DLL that ships with such programs should probably be used instead, as they will probably be newer and may support bells and whistles needed by the commercial application. I'll note any "commercial-only" features in the API documentation.

DSOUND WIN.INI Entries

DSOUND.DLL reads certain defaults from the WIN.INI file when the library is initialized. These are stored in the section tagged [DSOUND]. Values are read from WIN.INI only when the library is loaded, so later changes made to WIN.INI will not effect DSOUND.DLL. It is possible to update most of the current internal values through function calls described later. There is no DSOUND API support for changing the WIN.INI settings; use the Windows WriteProfileString function for this.

The DSound Control Panel can be used by the user to change the WIN.INI values and the run-time internal values, and most settings will be changed this way. The only internal value I can see a normal application needing to change is the global volume, to change the playback volume temporarily for an application. See Get/SetVolume below.

In the Aristosoft version of DSOUND.DLL 3.0, no WIN.INI settings are required for proper operation. However, the following delay settings may need to be altered.

These are the delay parameters in the [DSOUND] section of WIN.INI:

```
RDelayValue=nn
SDelayValue=nn
EDelayValue=nn
```

(nn is a positive integer).

These three entries set the delay value for sound playback in Real, Standard, and Enhanced modes, respectively. The delay value is mainly used in Enhanced mode to pace the playback of sound samples. A delay value of 0 (DS_TIMER in DSOUND.H) instructs DSOUND.DLL to use the built-in timer to pace playback; this is the default behavior for Real and Standard mode in Windows 3.0. (3.1 Beta I requires a non-zero delay for all modes, and the timer-driven code will probably disappear in a future version of the .DLL). The built-in timer cannot be used in Enhanced mode, though, so a delay value is needed. If the value of EDelayValue is 65535 (DS_AUTO in DSOUND.H) or the WIN.INI entry is missing, the .DLL will choose an appropriate delay value when the library is initialized. This is quite accurate (but takes a few seconds when the library is loaded). (As this code was develped under contract from Aristosoft, it won't appear in any shareware version for a year.) Any other value will be used as an explicit delay value. Note that changing the delay to 0 with SetDelayValue while in Enhanced mode or while running Windows 3.1 is bad. Don't do it. The control panel will warn you if you try to set a delay value to 0 in a mode this will probably cause crashes in. Similarly, you'll get a warning beep if the library is loaded with a suspicious 0 delay value.

The Sound Source can pace its own playback, so the delay value is ignored for this driver.

Under some circumstances, it may be necessary to use a non-zero delay value for Standard or Real mode, such as in Beta I of Win 3.1. Similarly, it may be necessary to bypass the auto-delay calculation if it sets the delay to the wrong thing. Both these can be done with the DSOUND Control Panel, or by writing the appropriate values to WIN.INI.

In the shareware version of the .DLL (and probably in future commercial products), there is no need for delays. The delay settings are ignored. Let me know if this implementation works.

Other WIN.INI parameters:

```
Volume=nn
Shift=nn
```

These store the global volume and shift values, described below. The default is 10 for volume and 0 for shift. Both can be changed from the control panel. See PlaySound below for a description of the Shift parameter.

```
Port=nn
SoundDev=nn
```

These allow for determining what hardware to use for playing the sound. Currently, the internal speaker is DS_INTERNAL (0), the Disney Sound Source is DS_SSOURCE (1), and the Sound Blaster is DS_SBLASTER (2). The Port setting will be zero for the internal speaker, and normally will be the base address of the LPT port for the Sound Source. It will be the base I/O port for the Sound Blaster (normally 220H) if this device is chosen. The value should (must!) be specified in decimal. Note that the delay value may need to be changed for each device, since the drivers are not equally fast (or slow).

For the Sound Source, LPTREDIR.COM must be run from DOS before Windows is run. This program does a few useful things. It prevents Windows from virtualizing the parallel port (and thus allowing the I/O calls to the hardware to be fast enough), and it provides the control panel with the I/O address of the port. The interface to LPTREDIR.COM is documented later.

```
KeyCheck=nnn
ESCKey=nn
```

These settings provide support for aborting sound playback by pressing the ESC key. Normally this is disabled because it degrades the sound quality. To enable it, set KeyCheck to a positive value that represents the number of samples to play between key checks, normally about 4000. Setting this to 0 disables such checking. ESCKey should be the scan code for the ESC key on your keyboard; it defaults to 0. You will hear clicks in the playback when key checking is enabled.

```
NoInts=[0|1]
```

Defaulting to 1, setting this value to 0 will allow interrupts to occur during sound playback. This causes a loss in quality, but will prevent interrupt-driven programs from losing data.

The DSOUND.DLL API

The DSound API allows C, Turbo Pascal/Windows, Visual Basic, Actor, and most macro languages to include sound support with minimal effort. In many cases, as little as 1 or 2 lines of code are needed. All functions are exported from DSOUND.DLL with the Pascal calling convention. All functions are prototyped in DSOUND.H (for C) and DSOUND.TXT (for Visual Basic and Basic-like macro languages). All references to DSOUND.H below are probably just as valid for DSOUND.TXT. Headers for other languages can be created from these two files in a straightforward manner.

What follows is the documentation for all public functions in DSOUND.DLL. The version of the earliest version of DSOUND.DLL that supported the function is also given (note that some versions were only used internally). Programs shound use GetDSoundVersion to make sure that all called functions are supported.

PlaySound [1.0]

The central function in DSOUND.DLL is PlaySound. It is mainly of use to C users, as the data structures it requires are not easily used in Visual Basic. However, for playing "computed" sound sampled, such as sine waves, it may be useful.

```
void FAR PASCAL PlaySound(char huge *lpSound, long dSize,
        unsigned uFrequency,  unsigned uSampleType, unsigned uVolume, unsigned
```

uShift)

lpSound is a long pointer to a chunk of memory with the samples in it, one byte each. This should be locked. GlobalAlloc/GlobalLock work fine. Page-locking in Enhanced mode, though not needed, will prevent the sound from stuttering as the sample bytes are paged into memory. lpSound is a char huge *; samples can be much longer than 64K, up to the maximum size GlobalAlloc'able under the current mode (1 Mb in Standard mode, 64 Mb in Enhanced mode). You can play sounds that are larger than available memory by allocating a block of memory and reading in the sound piece by piece, though there will be slight pauses between the pieces. PlaySoundFile does exactly this. Note that for most output devices, interrupts will be off while the sound plays. The time will be correct, but other interrupt-driven I/O will suffer.

dSize is an unsigned long that tells how many samples there are in lpSound. Currently, this should be the size of the sound data in bytes, but in the future this may not be so. Be accurate; Windows complains if DSOUND.DLL accesses memory you don't own.

uFrequency is the desired playback frequency. Currently 22 KHz, 11KHz, 7.33 KHz, and 5.5 KHz are supported. However, to be open-ended about this all, all frequencies are "valid," and the nearest one is used. That is, 9 KHz will play at 11 KHz, and 44 KHz will play at 22 KHz. Most sounds are recorded at 11 KHz. Note that the actual playback frequency may not be what is specified; samples may be skipped or double-played to compensate. This parameter tells the driver what rate the sample should be played back at. In fact, currently all samples are played at 22 KHz; slower sampling rates are supported by run-time smoothing of intermediate samples. (Ok, I lied. The Sound Source plays at 7 KHz. All kinds of nasty things are done to play the supported rates. I had thought it was playing at 22 KHz; it was just dropping samples...)

uSampleType is a code for the sample size. Currently only 8-bit linear is supported, which has the code 0. This parameter is currently ignored, but future versions may support 16-bit sample sizes, log scales, compression, and such. In other words, for future compatibility, pass 0.

uVolume adjusts the dynamics of the sound before playing. Basically, the amplitude of the sound centered around a middlepoint is expanded/compressed by the factor (uVolume/10). Thus, a volume of 5 is about half as loud, and 20 is twice as loud. In theory, that is--the PC speaker is not all that dynamic. Nor is the Sound Source. The 'Blaster is, but since its driver was hacked together from parts of the other drivers, I treat it the same. Best to use the physical volume control. Note that this value will be adjusted according to the global volume (Get/SetVolume) similarly. If the global volume is 5, and you pass 20 into PlaySound, the actual playback volume will be 10. Basically, the uVolume parameter should be used to make different sounds have the same apparent volume when played at the same global volume, while the global volume should be used to adjust the user playback level.

uShift is an amount added to each sample before being processed (but after the volume has been accounted for). This wouldn't be useful on a real sound chip, but it allows for compensating for different speaker hardware in this hack. 3-5 seem to work best; 4 is the default SOUNDER.EXE uses. Note that there is a global shift value as well (Get/SetShift). As with volume, the uShift should be used to make a specific sound sound right, while the global shift should be used to compensate for weird hardware. Almost all sounds require a shift value of 4, but those that are not well centered (the waveform isn't centered around 128) will need to be shifted. This parameter may not be used for all hardware output devices. Also, in hindsight, it should probably default to 0 for all files, with the driver compensating for its ideosyncracies, but c'est la vie! My first API design blunder...

GetDSoundVersion [1.0]

int FAR PASCAL GetDSoundVersion()

This function returns the version of DSOUND.DLL as an integer whose upper byte is the major revision and whose lower byte is the minor revision. The version should be checked before further calls to DSOUND.DLL are made to make sure that all functionality is supported.

Note: Because of a bug in Winword, it returns an error if you call a function with no parameters, even though there should be no error. Because of this, GetDSoundVersionD is also exported; it takes a dummy integer parameters that is ignored. Other Winword-compatible functions will appear later... Does anyone know why no Windows function takes no parameters?

AboutDSound [3.0]

void FAR PASCAL AboutDSound()

This function displays a dialog box that tells about DSOUND.DLL and its sub-drivers. The Version... button in the control panel calls this function. As above, AboutDSoundD should be used from within Winword macros; the dummy integer parameter is ignored.

SetDelayValue / GetDelayValue [2.0]

void FAR PASCAL SetDelayValue(int d)
int FAR PASCAL GetDelayValue()

These set or get the current delay value used internally. The WIN.INI setting is not affected. In Enhanced mode or under Windows 3.1 or "Windows-compatible platforms" of the future, setting the delay to 0 will most likely cause a crash. There's a GetDelayValueD as well. In future implementations, the semantics of this parameter may change.

SetVolume / GetVolume [2.0]

void FAR PASCAL SetVolume(int v)
int FAR PASCAL GetVolume()

These change/retrieve the current global volume for all sounds being played. The WIN.INI setting is not affected. If you plan on changing the volume, you should get it first and save it, returning it to its former state as soon as possible. GetVolumeD exists.

SetShiftValue / GetShiftValue [2.50]

```
void FAR PASCAL SetShiftValue(int s)
int FAR PASCAL GetShiftValue()
```

Some hardware may need a shift value other than the 4 that most sounds assume. I can't see why you'd need it, but GetShiftValueD is there.

SetSoundPort / GetSoundPort [3.0]

```
void FAR PASCAL SetSoundPort(int p)
int FAR PASCAL GetSoundPort()
```

Allows setting the I/O port used by the sound driver, if multiple ports are supported. This is generally of use only to the control panel. Probably best to leave it alone if you don't know what you're doing. As of version 3.0, only the Disney Sound Source driver uses this setting. Yes, why not: GetSoundPortD.

SetSoundDevice / GetSoundDevice [3.0]

```
void FAR PASCAL SetSoundDevice(int d)
int FAR PASCAL GetSoundDevice()
```

Allows for setting the driver to play sounds on. Currently the PC speaker, the Disney Sound Source, and the Sound Blaster are supported. DSOUND.H defines constants for these drivers. These calls are probably most useful to the control panel, but an application may decide to alter playback behavior based on the selected driver. Yup: GetSoundDeviceD.

UpdateClock [3.0]

```
BOOL FAR PASCAL UpdateClock(BOOL f)
```

Since many drivers run without interrupts, the PC speaker will fall behind. To compensate, with such drivers DSOUND.DLL will adjust the internal clock based on the sample size and frequency. This involves calling DOS, which under some circumstances may cause problems (such as when calling from a DOS critical error handler). Calling the function with FALSE will disable such updating. TRUE turns it back on. The return value is the previous setting. The time is updated in PlaySound and all functions that call PlaySound internally (PlaySoundxxx). Don't worry about accumulating error; when the flag is FALSE, the lost time is accumulated and reflected in the next updating of the clock. This function should rarely be needed; it was included to avoid a problem with playing sounds during disk errors in Wired for Sound, but is in the shareware version as well.

FindBestDelay [2.50]

```
int FAR PASCAL FindBestDelay()
```

This function is currently only useful in the version of DSOUND.DLL which ships with Aristosoft's Wired for Sound, as it was developed under an exclusive agreement with Aristosoft. In fact, it's pretty much useless to any program other than a control panel replacement. It calculates the best delay value for Enhanced mode (and Standard mode in Windows 3.1 Beta I). The delay value is returned, but the current delay value is not changed; use SetDelayValue() to change it. This call is really useful only in Enhanced mode, as the timer is more accurate and can be used in Real and Standard mode. Again, this call is mostly important to the control panel, as auto-delay sensing is implemented within DSOUND.DLL.

Note: As there should be no reason to call this from a Winword macro, there is no "dummy" equivalent. Thus, Winword will return a bogus error if this function is called from a macro.

PlaySoundFile [2.80]

```
int FAR PASCAL PlaySoundFile(LPSTR filename, int type,
                unsigned uFrequency, unsigned uSampleSize,
                unsigned uVolume, unsigned uShift)
```

This call will load and play the sound file specified by filename, if possible. Three file types are supported: .SOU, Sounder 1.0 .SND, and SoundTool .SND. The file type can be specified in the type parameter; if it is DS_GUESS, the file's type will be auto-detected as per below. The remaining four parameters are useful only if the sound file is of type .SOU; they are ignored for the other types since the parameters are in the file. The sound file need not be able to fit in memory; it will be read and played in chunks if necessary. [Note: there's a bug when sometimes this process fails and returns a bogus memory error. Probably won't be fixed....] Values for type and return values are defined in DSOUND.H. File formats are discussed later.

SoundFileType [2.80]

```
int FAR PASCAL SoundFileType(LPSTR filename)
```

Returns the type of sound file the specified file is (as defined in DSOUND.H). DS_CANTDETECT is returned if the file's type cannot be determined. SoundTool .SND files (DS_MHSND) are determined by looking for "SOUND" at the beginning of the file. If the file's extension is .SND but it isn't a SoundTool .SND file, it's assumed to be a Sounder 1.0 .SND file. If the extension is .SOU, it's assumed to be a .SOU file.

LoadSoundImage [2.80]

```
GLOBALHANDLE FAR PASCAL LoadSoundImage(LPSTR filename,
                int type, unsigned uFrequency,
                unsigned uSampleSize, unsigned uVolume,
                unsigned uShift)
```

This function loads a sound file into memory but does not play it. The memory sound image is in the Clipboard format as defined in SoundTool and by the SNDHeader structure in DSOUND.H and DSOUND.TXT. What is returned is an unlocked global handle. The parameters are as in PlaySoundFile. Unlike PlaySoundFile, though, the sound must fit entirely within memory. For Visual Basic and macro languages, the return value is simply an integer. Note that the sound should be freed as soon as possible to reclaim its memory. To do so, call the GlobalFree function (exported by the Kernel) with the value returned from LoadSoundImage.

PlaySoundImage [2.80]

```
int FAR PASCAL PlaySoundImage(GLOBALHANDLE hSound)
```

This call plays the sound memory image hSound. hSound need not be locked when this is called. This handle will usually be returned from LoadSoundImage. The memory image is assumed to be in the structure defined by SNDHeader. See DSOUND.H for possible error values.

PlaySoundResource [3.0]

```
int FAR PASCAL PlaySoundResource(HANDLE hSound)
```

This function operates exactly like PlaySoundImage, except that it uses a handle to a resource in the SND format (the SoundTool .SND format without the "SOUND^Z" header) obtained with calls to FindResource and LoadResource. The resource should be freed with FreeResource when it's not needed. Since few macro languages allow using resources, this is of little use to non-C and Pascal users.

_llread / _llwrite [2.8]

```
long FAR PASCAL _llread(int file, char huge *buf,
        long length)
long FAR PASCAL _llwrite(int file, char huge *buf,
        long length)
```

These have nothing to do with sound, except that they'll make reading in large sound files (> 64K) much easier. They're identical to _lread and _lwrite, except that a long is used for the length and return value, the buffer can be greater than 64K in size, and the read/write can cross segment boundaries.

Linking to DSOUND.DLL

To use DSOUND.DLL in a C program by linking implicitly, DSOUND.H should be included in your source files that access it and either DSOUND.LIB should be linked with your files or you should import the functions in your .DEF file. The .DEF file of DSOUND.DLL is included for copying the ordinals, if you don't have EXEHDR. As I eventually learned from Martin Hepperle, author of SoundTool, using import libraries and IMPORTed functions at the same time is bad!

For Visual Basic and Basic-like macro languages, the declares in DSOUND.TXT perform the necessary linking. They can be used as models for other programming languages.

File and Clipboard Formats

The format of the Sounder 1.0 .SND file is quite simple; the file has the following words, followed by the sample bytes:

word 0	Sample size code (see uSampleSize above)
word 1	Frequency to play back at (see uFrequency above)
word 2	Volume to play at (see uVolume above)
word 3	Shift (see uShift above)
word 4..n	Actual sample bytes to be played

The SoundTool .SND format is better, and should be used if possible. Wired for Sound's sound files are in this format. It closely matches the memory-image formats and resource format. Only files have the 6-byte SOUND^Z header; resources and memory images begin with the hGSound, which is currently ignored. In fact, I'm not sure why it's there at all, except that's how SoundTool defined things. Perhaps for a linked-list implementation of sounds...?

```
char szMagic[6] = { "S","O","U","N","D", 0x1a }
GLOBALHANDLE hGSound;   /* not used */
DWORD dwBytes;      /* length of complete sample */
DWORD dwStart;      /* first byte to play from sample */
DWORD dwStop;       /* first byte NOT to play from sample */
WORD wFreq; /* frequency */
WORD wSampleSize;
WORD wVolume;
WORD wShift;
char szName[96];      /* name of sound */
Sound sample follows this header.
```

This format can also be used for clipboard transfers under the CF_SOUND type.

A third "format" is the .SOU format, which is just a stream of bytes. Since it is necessary to specify the playback parameters manually, this is impractical for most uses. It's included since it's the format other software uses (i.e. reMac), and was supported in Sounder 1.0. A non-compressed Sound Blaster .VOC file can be read in as a .SOU file; there will be a bit of garbage at the beginning (the .VOC header). There's a VOC2SND program in the public domain, I believe.

LPTREDIR

LPTREDIR.COM needs to be run before running Windows in order to use the Sound Source. The command line format is simply:

```
LPTREDIR x
```

where x is the LPT port the Sound Source is connected to: 1, 2, or 3. All other characters on the command line are ignored. A port number must be given. LPTREDIR can be loaded only once. The port cannot already be redirected (by a network.)

Once installed, you can call LPTREDIR via INT 17H:

If AH = 0c7h, the following are returned if LPTREDIR is installed:

```
AX= 3456h
BX=<I/O port address>
CL=<LPT number of port>
```

Not very exciting. The control panel uses this information to see if LPTREDIR is installed, and to get information for the Port= line in WIN.INI. Note that DSOUND.DLL doesn't call INT 17h to get the port address. There's a reson for this...

What LPTREDIR does is make Windows think the desired parallel port is under network redirection. This makes Windows print through the BIOS when printing to a connected printer, which isn't too bad. But more, in Enhanced mode (and seemingly in Standard mode in 3.1...!) it prevents Windows from virtualizing the printer port, allowing the driver to be fast enough. This is done in the "standard" way; after LPTREDIR is loaded, the port address in the BIOS data area is changed to 0001. This means "Yes, there is a LPTx:, but don't use the hardware." Most programs use this protocol. Run DEBUG and d 40:0 before and after LPTREDIR is loaded. When the BIOS is called, LPTREDIR swaps back in the port address, calls the BIOS, and reverts is before IRETing.