Plug-In Development Kit (PIDK) for DirectX Audio Plug-Ins

Release 5

Sonic Foundry, Inc.

Last Updated: July 12, 1998

End-User License Agreement

Sonic Foundry Plug-In Development Kit

You have a non-exclusive, royalty-free right to use the information and source code contained in this Plug-In Development Kit (PIDK) provided you adhere to the following restrictions:

1. You may distribute object code based on the examples in the PIDK, provided that significant changes have been incorporated into the functionality of the example plug-ins.

2. You may *not* distribute source code, or any other component of the PIDK, in *any* form without express written permission from Sonic Foundry.

Disclaimer of Warranty

NO WARRANTIES. THE SOFTWARE PRODUCT IS PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, SONIC FOUNDRY AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY AGAINST INFRINGEMENT, WITH REGARD TO THE SOFTWARE PRODUCT. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION.

CUSTOMER REMEDIES. SONIC FOUNDRY'S ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL NOT EXCEED THE PRICE PAID FOR THE SOFTWARE.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL SONIC FOUNDRY OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SONIC FOUNDRY PRODUCT, EVEN IF SONIC FOUNDRY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES/JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

Contents

The contents of this manual are as follows:

Introduction

The Sonic Foundry Plug-In Development Kit (PIDK) for DirectX Audio Plug-Ins is intended as a supplement to Microsoft's DirectX Media Streaming Services (DMSS), formally known as ActiveMovie, Software Development Kit (SDK). The documentation and samples contained in this PIDK are designed to expand on the existing DirectX SDK in the area of digital audio signal processing.

The DMSS SDK (available from Microsoft) theoretically provides all information needed to develop DirectX Audio Plug-Ins. However, the information related to digital audio "filters" is sparse, and might be difficult to assimilate for programmers who are new to the DirectX Media Streaming Services.

This PIDK does not define a new standard for plug-ins, it simply clarifies Microsoft's specification in relation to digital audio. Plug-ins written with the information contained in this PIDK should work in any DMSS-compatible application (assuming audio transform filters are supported).

Please note that Sonic Foundry uses the term "Plug-In" to refer to what Microsoft calls "Filters." End-users are more aware of the term "Plug-In," while the word has the added benefit of being less ambiguous than "Filter."

Background

DirectX Audio Plug-Ins are derived from Microsoft's DirectX Media Streaming Services (DMSS), formally known as ActiveMovie, specification, which is a component of Microsoft's ever-expanding DirectX technologies. DMSS was originally designed as a new media-streaming architecture that would deliver high-quality video playback, replacing Video for Windows. However, the underlying technology exposes a flexible and highly extensible set of interfaces that can easily be used as a standard plug-in architecture for video- and audio-related applications.

Based on these capabilities, Sonic Foundry has chosen DMSS as the foundation for developing plug-ins for all Sonic Foundry products.

Requirements

The Sonic Foundry PIDK requires the following components to develop DirectX Audio Plug-Ins:

- Windows 95, Windows 98 or Windows NT 4.0 (or later)
- Visual C++ 5.0 or later
- DirectShow (DirectX Media 5.2a SDK or later)

If you need the latest DirectX Media SDK or if you are not familiar with Microsoft's DirectX Media Streaming Services technology, you can peruse Microsoft's DirectX information from the following URL:

http://www.microsoft.com/directx

Contacting Sonic Foundry

You may contact us via any of the following methods:

Mailing Address

754 Williamson Street
Madison, WI 53703
USA

Telephone

608.256.3133 (Sales and Business)

Fax

608.256.7300

Web

http://www.sonicfoundry.com

E-Mail (for PIDK Purposes)

pidk@sonicfoundry.com

Technical Support

Sonic Foundry provides no official technical support for the PIDK due to limited resources. Questions sent to **pidk@sonicfoundry.com** may get answered by Sonic Foundry's development team, if time permits and the question is not thoroughly answered in this manual, sample code, or Microsoft's documentation. If we are unable to answer the question, we will do our best to respond accordingly.

Questions about DirectX Media Streaming Services (DMSS) in general should be directed to Microsoft's developer support.

Please do not call with questions regarding the PIDK unless a member of our development team specifically requests a call. Email is the fastest and least intrusive (most accepted) method of reaching Sonic Foundry's developers.

Feedback

Sonic Foundry is very interested in all comments and suggestions related to the PIDK. If you notice a bug in the sample code or incorrect information in any documentation, we would appreciate it if you would notify us of the error. Due to limited resources, however, please do not telephone us with this information. Rather, please send email to:

pidk@sonicfoundry.com

Please do not assume that the error is too minor, too obvious, or has already been reported.

Plug-In Development Kit Overview

The Sonic Foundry Plug-In Development Kit (PIDK) for DirectX Audio Plug-Ins is a supplement to Microsoft's DirectX Media Streaming Services (DMSS), formerly known as ActiveMovie, Software Development Kit (SDK), **not** a replacement. The focus of this PIDK is digital audio signal processing and, with the sample code, it shows how to solve common audio signal-processing problems using the DMSS SDK.

No attempt is made to republish information that is contained in Microsoft's documentation. If you are unfamiliar with DMSS, the Component Object Model (COM), C/C++, or you do not know how to empty the Windows 95 Recycle Bin, then you are not ready for this development kit. **The PIDK documentation and sample code assumes that you have read and understand the basics of DMSS.**

The Frequently Asked Questions section of the DMSS SDK Help file is a fantastic starting point for learning about DMSS.

The learning curve on DMSS can be steep, but fortunately, if all you want to do is create audio transform plug-ins, a fairly good template is available with the DMSS SDK called Gargle. Gargle does have a few bugs, and while Microsoft knows about these bugs and intends to fix them, they have not yet released an SDK that contains the fixes. In the meantime, Sonic Foundry has included a complete description of all known Gargle bugs and how to fix them in the "Known Bugs and Limitations of DirectShow" section of this document.

PIDK Contents

The Sonic Foundry PIDK consists mainly of sample code demonstrating how to write various transform-related audio signal-processing plug-ins. These examples were derived from Microsoft's Gargle example (though we fixed the bugs), with extensive changes to show how common audio signal-processing problems can be solved using DMSS.

The examples in this PIDK are not intended to teach digital audio signal processing techniques. These examples are toys. The quality of the signal processing is a joke. If this isn't obvious, you might want to consider alternate employment. These are demonstrations of using DMSS as a digital audio plug-in architecture, nothing more.

Sonic Foundry has included the entire source code for all examples in this PIDK. There are no proprietary DLLs or libraries required to use these examples (other than Microsoft's DMSS components), nor are you required to ship any Sonic Foundry components. It's all here. You are welcome to modify them in any manner you see fit, provided you adhere to the End-User License Agreement (EULA) contained with this PIDK.

To quickly summarize the EULA (but not replace), you **may** distribute object code based on these examples (with significant changes) royalty-free. You may **not** distribute the source code in any form without written permission from Sonic Foundry. If you have additional questions, please refer to the End-User License Agreement on page 3.

Pros and Cons of DirectX Media Streaming Services (DMSS)

It is important to understand that DMSS has limitations, many of which are discussed throughout this document, and is not an end-all solution to the proprietary plug-in architecture problem. However, DMSS is capable of solving a wide variety of common plug-in problems, and can be extended with custom interfaces. This makes it an excellent *foundation* for a standard plug-in architecture.

Sonic Foundry is committed to using DMSS as our future plug-in architecture. This architecture is powerful, flexible, and reasonably fast, but most importantly, it is *not* proprietary to any one company outside of Microsoft (like Windows itself). DMSS is becoming an integral part of the Operating System, which makes it a very convenient long-term solution.

One very advantageous aspect of DMSS is the fact that it was designed from the beginning to take advantage of multi-threading and multi-processor machines (on Windows NT).

DMSS plug-ins are not required to be software only. They can take advantage of any acceleration hardware they desire (DSP boards, for example).

Floating Point for 24-Bit Audio

Microsoft, in conjunction with Sonic Foundry and other companies, has defined a new wave format data type for working with greater than 16-bit PCM data. This new format uses the Institute of Electrical and Electronics Engineers (IEEE) standard 32-bit floats (or 64-bit doubles), rather than 16-bit integers for each audio sample. To encourage the use of this new data type, all example plug-ins contained in this PIDK support 32-bit float processing.

Microsoft has defined the new data type as follows. This is defined in the SFIFACE.H file, and should appear in future versions of Microsoft's Win32 SDK:

```
#define WAVE_FORMAT_IEEE_FLOAT       3
```

This new data type is used like this:

```
LPWAVEFORMATEX pwfx;

pwfx->wFormatTag      = WAVE_FORMAT_IEEE_FLOAT;
pwfx->nChannels       = <number of channels>;
pwfx->nSamplesPerSec  = <sample rate>;
pwfx->wBitsPerSample  = 32; // or 64 for doubles
pwfx->nBlockAlign     = pwfx->nChannels * (pwfx->wBitsPerSample / 8);
pwfx->nAvgBytesPerSec = pwfx->nBlockAlign * pwfx->nSamplesPerSec;
pwfx->cbSize   = 0;
```

The above example is for 32-bit floats. 64-bit doubles (probably overkill, but certainly possible) can be supported by setting `wBitsPerSample` to 64. Note that while there is only one defined format for 32-bit IEEE floats, there are actually two variants for 64-bit floats defined by the IEEE standards committee. It is best to assume that the 64-bit float format is not portable.

There are two very basic rules when using this new data type:

- The floats (or doubles) are normalized to the range of -1.0 to 1.0
- Overdriven samples are valid and must be bound by the playback (rendering) device

By doing this, more than 24 bits of precision (*) can be maintained, even with an overdriven signal, up to, but not including, a value of plus or minus 2.0. Precision is lost beyond this point, but hopefully that is a result of the user applying too much gain. If you have a chain of plug-ins, one plug-in can overdrive the signal and the next plug-in may cause a gain reduction, bringing it back to a normal range. This obviously has many benefits.

This new format, when used properly, provides very high performance on Pentium [Pro] and RISC processors, while preserving more than 24 bits of precision. In addition, the added benefit of using floating point for DSP routines can make designing and developing algorithms much simpler than using fixed point. Significant performance gains are achieved by maintaining floating point values without converting back and forth to integers. Remember, plug-ins will frequently be chained.

Converting to and from the floating point data type is done as follows:

Data Type:	Conversion To:	Conversion From:
8-bit PCM	(Sample ^ 0x80) * 128.0	(Sample / 128.0) ^ 0x80
16-bit PCM	Sample * 32768.0	Sample / 32768.0
24-bit PCM	Sample * 8388608.0	Sample / 8388608.0

When converting from a floating point data type to a PCM (integer) data type, you must bound the result. The following example converts a float data type sample to a 16-bit PCM value:

```
float   Sample  = 0.123456;
long    lSample = (long)(Sample * 32768.0)
short   sSample;

if (lSample > 32767)
        sSample = 32767;
else if (lSample < -32768)
        sSample = -32768;
else
        sSample = (short)lSample;
```

The IEEE float data type is also fully supported by Microsoft's RIFF .WAV file format. Files should be written exactly like compressed audio files, even though the data is not actually compressed. Microsoft provides ample information on compressed audio files with the Audio Compression Manager (ACM) documentation. In addition, there is an example called ACMAPP, which is widely available on most Win32 SDKs provided by Microsoft.

<Pause to mount soapbox>

Critics of using floating point should probably spend a little time profiling some non-trivial DSP code, especially on a Pentium Pro (for this, a small amount of competence using floating point is assumed). Gone are the days of the 486 and Windows 3.1 (16-bit code), so it's time to reevaluate some outdated beliefs. Also, all RISC platforms (Alpha, MIPS and PowerPC processors) perform extremely well with floating point. Think about your target market: if you're using DirectX Media Streaming Services, your target is, by default, Pentium and better processors.

Admittedly, simple processing algorithms (and a few special cases) tend to perform better using integer (fixed-point) math. However, these gains can quickly be consumed if you are converting back and forth between floating point and integer values (think about a chain of plug-ins trying to maintain 24 bits of precision). For these cases, we encourage writing two versions of the DSP routine: one for floating point connections, and one for 16-bit PCM connections. This helps to maintain the highest throughput possible when chaining plug-ins.

Whatever you do, don't forget this rule: **always profile code before making assumptions about performance**. If the time required to convert back and forth between floating point and integer values is outweighed by performance increases during processing, then by all means, do so. However, before making this assumption, *profile, profile, profile*.

Looking to the future, we expect plug-ins that work with floating point to consistently equal or outperform plug-ins that work on integer (fixed-point) samples, especially when trying to maintain more than 16 bits of precision.

<Pause to step down from soapbox>

Sonic Foundry products fully support connections of 16-bit PCM and 32-bit float data types. With Sound Forge, you can switch between these connection types at any time by right-clicking on the processing bit depth indicator in the lower right corner of the property page.

Note: A small amount of overhead is added by Sound Forge when processing 32-bit floats due to its current storage implementation (16-bit PCM). This overhead is reflected in the CPU usage meter (which is only an approximation), and should be considered when comparing performance. Always use your own profiling code.

Included Example Plug-Ins

Below is a list of the example plug-ins included in this PIDK, with a brief description of what the example is intended to demonstrate.

SFInvert
Demonstrates the most primitive in-place transform possible. This example modifies the audio data in-place and does not lengthen, shorten, or change the media type. There is no user interface (e.g., no property pages). This example simply inverts the sign of each sample (positive samples become negative, and vice versa) and is a good starting point for very simple processors that require no user interface.

SFGargle
Demonstrates an in-place transform with a very simple property page. This example modifies the audio data in-place and does not lengthen, shorten, or change the media type. SFGargle is a more robust version of the Gargle example that Microsoft supplies on the ActiveMovie SDK. This example is a good starting point for most signal processing that does not change the length of the data (compressor/limiter, EQ, gates, distortion, vibrato, pitch change preserving duration, etc.).

SFEcho
Demonstrates an in-place transform that extends the data (adds a tail) during the end of stream processing. It also has a simple property page. This example modifies the audio data in-place and only lengthens the data through the end of stream processing. The media type is not changed. This example is a good starting point for reverb and echo type effects, where a tail for decaying signals is necessary.

SFLength
Demonstrates a non-in-place transform that either lengthens or shortens the data. It also has a simple property page. This example copies audio data from an input buffer to a shorter or longer output buffer; the effect is a (very poor quality) pitch change without preserving duration. The media type is not changed. This example is a good starting point for effects that lengthen or shorten the duration of the data (time compress/expand, pitch change without preserving duration, etc.).

SFMix
Demonstrates a non-in-place transform with a media type change. It also has a simple property page. This example changes a stereo input to a mono or stereo output with independent left and right channel mixing level controls. This example is a good starting point for effects that need to change the number of channels from input to output (stereo to mono converters, 5.1 to stereo converters, etc.).

SFPan
Demonstrates a non-in-place transform with a media type change. It also has a simple property page. This example changes a mono or stereo input to a stereo output with left and right channel balance control. This example is a good starting point for effects that need to change the number of channels from input to output (mono to stereo converters, panning, 3D audio processors, mono in/stereo out reverbs, etc.).

SFStress

Demonstrates a non-in-place transform that extends the data (adds a tail) during the end of stream processing. This plug-in will function exactly like SFEcho, but also adds several stress case situations that are designed to test host environment capabilities.

Three sliders have been added for this testing: Buffer, Load, and Preset. The *Buffer* slider controls the size of the buffer that is required to be full before processing can start. This simulates FFT or windowing-based effects that require a given amount of data to be processed in a chunk that is larger than what is normally delivered. Similarly, it packages the output into a buffer (in this case the same size as the input), and begins delivering data, utilizing as much of the output buffer as possible.

The *Load* slider simulates intensive processing. This will simply control the computational intensity that is done per sample during the actual signal-processing algorithm.

The *Preset* slider emulates large presets, and more importantly, presets that do not remain constant. It is fairly common to find environments that will not store a large preset, or will not store presets that change size from one another. Below the *Preset* slider are two text boxes. These boxes will display either FAIL or PASS for both read and write operations. These boxes detect if a read or write operation on the preset was successful; however, this is not a guarantee that the presets will work properly. Although the preset has read from or written to the persist stream, it is completely feasible for some other aspect to negate a successful read or write (i.e., a corrupt preset header).

SFStress should also be used to test host applications for support and functionality of multiple property pages. SFStress has three property pages. The first property page is a simply a sum of the second and third pages. The second page gives you signal processing controls similar to SFEcho, while the third page is of more importance while testing stress situations. All pages should be functional during real-time processing.

This SFStress example merely attempts to show some stress case situations that a host application should be able to support. It also demonstrates handling tail data in a large windowing manner (useful for FFT based processing). This example was not designed for best performance—it was designed to stress host applications.

What's New in the Plug-In Development Kit (PIDK) Release 5?

Below is a description of new features in Release 5 of the PIDK:

DirectX 5.2a Support

To keep the PIDK plug-ins current with the latest release of DirectX Media Streaming Services (DMSS), you will be able to compile with DirectX 5.2a Media Software Development Kit (SDK) only. ActiveMovie 1.0 is no longer directly supported. However, understanding the changes makes converting an example back to ActiveMovie 1.0 trivial.

Several important changes in the DirectX 5.2a Media SDK are relevant to the PIDK, and are important to understand:

1. Two base classes no longer have a pointer to a HRESULT as a parameter. These two classes include CBasePropertyPage, and CTransformFilter for non-in-place transforms.
2. AMovieDllRegisterServer and AMovieDllUnregisterServer have been rolled into a single boolean call to AMovieDllRegisterServer2(BOOL). This will require the g_Templates structure to be modified..
3. CFactoryTemplate g_Templates requires two new parameters to fully support the DirectX Media 5.2a registration process.

   By utilizing the AMovieDllRegisterServer2(BOOL) call, you will be registering the plug-in in accordance to the DirectX 5.2a SDK specification.

Directed Toward Visual C++ 5.0 Environment

This release of the PIDK has also been updated to follow menus and dialogs from Microsoft's Visual C++ 5.0 environment. All example setup procedures use Visual C++ 5.0 exclusively. This migration to Visual C++ 5.0 also eliminates the need for the OLECTLID.H header file in the SFPLUGIN.CPP file. The code in this file has been rolled into OLECTL.H. Also, the Visual C++ 5.0 compiler complained that the exports in the .DEF files were not explicitly PRIVATE.

TransformDispatchConnectNotify Function

We continually get questions like "How do I send a message to the UI (Property Page) from my signal processing code?" The TransformDispatchConnectNotify(..) function in sfplugin.cpp has been renamed to TransfomDispatchMessage(..). The functionality remains essentially the same, and has actually always served as a means to notify the UI of changes. However, this release of the PIDK has a better demonstration of how to comminicate between your DSP code and your UI. Be sure to look at SFPPAGE.CPP to see how the message is actually handled. We recommend that you use this feature with caution. Do not feel free to spew a million messages to the dialog. Be smart, collect information, and use TransformDispatchMessage(..) when necessary.

Force Dialog Update

It was found that the example plug-ins in previous releases of the PIDK did not update the Property Page after a user preset had been loaded in certain applications. We have provided a more robust solution to the problem by using the new TransformDispatchMessage(..) (see above for explanation) to force an update of the property page when a user preset has been loaded. However, a brief flash of the property page may occur when other update methods are simultaneously used.

Bug Fixes

The following is a list of important bug fixes in this release of the PIDK. However, this list is not necessarily conclusive. Additional modifications (i.e., spelling, punctuation) have been made, but are not necessary to add to this list.

1. A put_all(..) statement has been placed in TransformRead(..). Previously, a user preset was loaded directly into the member variables, and was not flipping the “dirty” bit while previewing.
2. #define FLOAT_SAMPLES_ONLY in Sfgargle would cause initialization of several variables to be skipped by a case statement. The fix in the SFPLUGIN.CPP file was to simply include a set of { }.

Writing a DirectX Audio Plug-In

Before you begin looking at the PIDK sample code, please make certain you can build and debug the Gargle example that ships with the DirectX Media Streaming Services (DMSS) Software Developers Kit (SDK). Until you can do this, this PIDK cannot help you. After successfully building and analyzing (stepping through) the Gargle example using Graph Editor, Sound Forge, or other DirectX environment, then you can build all of the PIDK examples, and get familiar with what they do.

This PIDK assumes you have installed the DMSS SDK to C:\DXMEDIA. Be sure to change this if you have chosen an alternate installation path. Also, be sure to put the SFIFACE.H header file (from the SFPIDK\INCLUDE folder) somewhere in your INCLUDE path. This header file is required by all of the Sonic Foundry example plug-ins.

You can use any Sonic Foundry application that supports DirectX Plug-Ins for self-registering the example plug-ins. Just drag and drop each of the .AX files (found in C:\DXMEDIA\BIN after building them) onto to Sound Forge or ACID and follow the instructions.

We have tried very hard to make these examples useful and flexible, while still minimizing and localizing the changes that must occur to make a new plug-in based on this code. These examples should be a good starting point. A lot of improvements can be made for efficiency, but we didn't want to be too tricky. These are generic examples, not brain teasers.

All of the generic DirectX Plug-In "glue" code has been put in the file SFPLUGIN.CPP, and in most cases, you will not need to modify this code.

Building With Visual C++ 5.0

This section describes how to create a project file within the Visual C++ 5.0 Developer Studio that can be used to develop DirectX Audio Plug-Ins. This, sadly, is more complicated than it should be. However, it can be done.

Note: the PIDK examples are designed to be edited completely with the Developer Studio (including all resources). This makes it very convenient for development, but may not be the best idea for large projects. We have given you all the code, so change whatever you need for your development environment.

The following example is for the SFPAN example.

Follow the steps below to create a project file within the Visual C++ 5.0 Developer Studio:

1. Select the **File** menu, followed by the **New** menu option.
2. Choose the *Project Tab.*
3. Select *Win32 Dynamic-Link Library.*
4. Verify that *Location* is the root of the PIDK (i.e., C:\SFPIDK).
5. Enter the project name (i.e., SFPAN), and click the **<OK>** button.
6. Select *Project.Add to project.Files.*
7. Add only the .CPP and .RC files (no header files).
8. Add the .DEF file.
9. Select the **Tools** menu, followed by the **Options** menu option.
10. Switch to the *Directories* page.
11. Add each of the following files to the Include Files list:

    C:\%DIRECTXSDKROOT%\INCLUDE

    C:\%DIRECTXSDKROOT%\CLASSES\BASE

    C:\SFPIDK\INCLUDE

12. Next, add the following to the Library Files list:

    C:\%DIRECTXSDKROOT%\LIB

13. At this time, you should be able to select *Build.Update All Dependencies* and get no errors. However, you will not be able to build yet!
14. Now select the **Projects** menu, followed by the **Settings** menu option.
15. Switch to the *C/C++* page.
16. Select *General*.
17. Add the following Preprocessor Definitions:

    INC_OLE2, STRICT

    Note: This is optional in the examples, since these defines are forced at the top of each .CPP file.

18. Select *Code Generation* and set the Run-Time Library to *Multithreaded using DLL* (the debug version is a good choice while developing)
19. Also on the *Code Generation* category, you must set the *Calling Convention* to __stdcall, or you will get link errors because of incorrectly prototyped functions in the Microsoft ActiveMovie base classes.

20. Switch to the *Link* page.
21. Set your *Output File Name* (i.e., C:\%DIRECTXSDKROOT%\BIN\DEBUG\SFPAN.AX).
22. Add the following libraries to the *Object/Library Modules*:

    STRMBASE.LIB
    VERSION.LIB

    Note: If you are using any "multimedia" API's, or the DirectShow 5.2a SDK, you will also need to add the following library:

    WINMM.LIB

23. If you want to bypass the DllMain leapfrog that SFPLUGIN.CPP uses (not critical), you can set the *Entry-Point Symbol* in the *Output* category to DllEntryPoint.

24. You should now be able to build and debug your plug-in using Developer Studio.

Note: It is highly recommended that you use the standard makefile for final builds.

Creating a New Plug-In

The first step in creating a new plug-in using the examples is to decide which example design most closely matches your signal processing requirements. Refer to the "Example Plug-Ins" list earlier in this manual for descriptions of each example. For example, if you are writing a Reverb, start with either the SFEcho or SFPan examples. If you are writing an EQ or Compressor, start with the SFGargle example.

The following example details how to convert the SFGargle example into a template for an EQ plug-in. The name of the new example plug-in will be **ZYZEQ**. These steps are essentially the same, regardless of which example plug-in you start with.

Note: If you are using the Developer Studio in Visual C++ to edit your files, make certain you use "Open as Text" for the MAKEFILE and .RC file when replacing names.

1. Create a new folder called ZYZEQ.
2. Copy all files from the SFGARGLE folder into this new folder.
3. Rename all SFGARGLE.* files to ZYZEQ.*
4. Edit the _DEPEND file, and replace all occurrences of SFGARGLE with ZYZEQ.
5. Edit the MAKEFILE file, and change the TARGET_NAME to ZYZEQ.
6. Edit the ZYZEQ.HPJ file, and change SFGARGLE.RTF to ZYZEQ.RTF.
7. Open each of the following files, and replace the "#include sfgargle.h" with "#include zyzeq.h":

   ZYZEQ.CPP
   ZYZEQ.RC (two occurrences)
   SFPLUGIN.CPP
   SFPPAGE.CPP

8. Open the RESOURCE.H file.
9. Replace the comment at the top that reads, "Used by SFGARGLE.RC", with "Used by ZYZEQ.RC". This keeps Developer Studio happy.
10. Open the ZYZEQ.H file.
11. Change SFPLUGIN_NAME and SFPLUGIN_HELP_FILE from using GARGLE to ZYZEQ.
12. **VERY IMPORTANT:** Generate four new GUIDs using GUIDGEN.EXE (comes with most Microsoft SDKs and Visual C++).
13. Use the above generated GUIDs to change the following defines in the ZYZEQ.H file:

    CLSID_SfPlugIn
    CLSID_SfPlugInPropPage
    CLSID_SfPlugInPropPage2
    IID_ISfPlugInProp

At this point, you should be able to build a completely new (and unique) plug-in, called ZYZEQ.AX. You can register this plug-in (by dragging and dropping it on a Sonic Foundry DirectX compatible application, or use REGSVR32.EXE), and run it without conflicting with the SFGargle example. Starting from this point, you can now start converting the code to implement your own signal processing. The following are the next recommended steps:

1. If you use the Visual C++ Developer Studio, create a new Project Workspace for your plug-in (see the "Building with Visual C++ 5.0" section earlier in this manual).
2. Edit the ZYZEQ.RC file (using Developer Studio's resource editor, if you wish) to reflect your plug-in and company names, copyrights, etc.
3. Edit the string resource for the plug-in name (IDS_PROPPAGE_1) that is displayed for the main property page.

At this point, your plug-in should be completely unique. Next, start modifying code and resources to implement your signal processing. The PIDK examples use unique qualifiers to make search and replace operations trivial (we are not trying to force a style preference). That is, you can search for SFGARGLE and replace it with ZYZEQ. This is also true for the class names that all begin with the SfPlugIn prefix. We have given you the entire source code, so change whatever you like. There is only one caveat: the more you change, the more difficult it will be to incorporate updates to the examples from Sonic Foundry. For this reason, you may want to leave the class names alone.

Adding Your Own Code

The following steps explain how to start adding your own code to the PIDK example:

1. Open the ZYZEQ.H file.
2. Add your DSP parameters to the TRANSFORM_PROPS structure (you can also delete the current SFGargle parameters, or leave them until later).
3. Update the PRESET_INITIALIZER (in the beginning, it's easiest to leave only the default preset, and then add the others when the code is closer to complete).
4. Update the GETPUT_SFPROP declarations in the ISfPlugInProp interface to reflect your TRANSFORM_PROPS structure.
5. Update the GETPUT_SFPROP declaration (this one is different from above!) in the CAudioTransform class to reflect your TRANSFORM_PROPS structure.
6. Update the put_all method of the CAudioTransform class to include bounding for all TRANSFORM_PROPS members. This is important for handling garbage presets that you might receive.
7. Add whatever 'cooked' members you need, if any, just after the m_fCooked member in the CAudioTransform class.
8. Now update the SFPPAGE.CPP file to incorporate the user interface for changing the transform properties.

Updating the Signal Processing

The only thing left to do (besides the help file) is update the signal processing itself (in the ZYZEQ.CPP file). The following is a list of the things to look for and change:

1. The constructor of CAudioTransform needs to initialize your CAudioTransform members to default values.
2. Change CAudioTransform::InitState and CAudioTransform::TransformCookProperties to initialize all cooked data (computed whenever a parameter changes).
3. **Note:** Regarding ::InitState in SFGargle, great care is taken to not rebuild the shape array unless it is absolutely necessary. This is highly recommended for anything that requires time consuming computations (such as initializing coefficients).
4. Change the processing loop for ::Transform16bit and ::TransformFloat.

That's it. Once you are ready to test your new plug-in, we recommend that you use Sonic Foundry's Sound Forge. It gives you flexible testing control (on-the-fly switching of formats, better real-time response, etc.), and handles property pages properly.

Finally, be sure to update the Help file (or create your own using the tools you are comfortable with). The Help identifiers are contained in the CONTEXT.H file, and need to be associated with all of your Property Page controls in the WM_HELP case of the ::OnReceiveMessage method in the SFPPAGE.CPP file.

Customization Defines

Each of the example plug-ins have customization #defines in their main header file. These defines can be used to change how the plug-in will build. However, you will still need to modify code to complete the functionality of the #defines (by searching for the define location throughout the source code). The following is a list of the customization defines, and a brief description of what they control:

SFPLUGIN_NAME
Specifies the user-visible name of the plug-in. Spaces and most punctuation options are allowed. This name will appear in menus and other portions of DMSS-compatible applications.

SFPLUGIN_HELP_FILE
Specifies the name of the Help file that the plug-in will use.

MULTIPLE_PROPERTY_PAGES
Define this to turn on the code that supports more than one property page per filter (rather than just one). You can search on MULTIPLE_PROPERTY_PAGES to find all of the places to touch to get two or more pages. It is not recommended to have too many property pages, as this can make using the plug-in tedious.

FLOAT_SAMPLES_SUPPORTED
Define this if you will supply a TransformFloat function that can handle buffers containing 32-bit float audio data.

FLOAT_SAMPLES_ONLY
Define this if, and only if, you also define FLOAT_SAMPLES_SUPPORTED, and you want SFPLUGIN.CPP to convert buffers of 16-bit PCM data to floats before calling your TransformFloat function. Unless this is defined, you will need to provide a Transform16bit function. Note that SFLENGTH, SFMIX and SFPAN currently do not support this define.

NEED_END_OF_STREAM
Define this if you wish to generate extra output data after all of the input data has been processed. Your TransformXXX function will be called with empty buffers for you to fill until you return S_FALSE.

NEED_CHANNELS_IN
If your plug-in requires a specific number of channels for input, you can specify the requirement with this define.

NEED_CHANNELS_OUT
If your plug-in requires a specific number of channels for output, you can specify the requirement with this define.

Combining Plug-Ins

If you intend to ship more than one plug-in as a package, it is most efficient (and convenient) to combine the plug-ins into a single .AX file. This is entirely possible, but the examples in this release of the PIDK are not set up for this by default.

One of the main incentives for combining your plug-ins is size. The base classes and associated baggage is not insubstantial. As an example, all of the PIDK examples except SFInvert compile into .AX files that are each larger than 50k in size!

The best approach to combining plug-ins is to simply link the static .LIB files (generated by the standard makefiles) into a single .AX file. To make this work, you'll need some "glue" code which defines custom instance creation functions (in the SFPLUGIN.CPP file). You will also need to create a master .RC file that includes all resources for each plug-in. Some of the code, specifically, the custom CreateInstance functions that can be called from the glue code, will need to be #ifdef'd, depending on whether it is being built as a stand alone or as a static .LIB. The DllRegisterServer and DllUnregisterServer functions will need to be #ifdef'd as well.

It's a little bit of work, but not impossible.

Testing and Debugging a DirectX Audio Plug-In

This section contains tips for testing and debugging a DirectX Audio Plug-In.

Audio Driver Bugs

A number of (well-known) audio card drivers contain bugs that can make testing your plug-ins very difficult. Many of these bugs show themselves as dropouts, clicks, pops, etc., while previewing in real-time. This section can help you recognize what is an audio driver bug versus a bug in your plug-in.

If you encounter audible glitches in Sound Forge while previewing in real-time, ***and*** the CPU usage meter is reading less than 100 %, then you may be running into a bug in your audio driver. This is usually only a problem in plug-ins that change the *length* of the audio data [for example, a stereo-to-mono conversion (SFMix), or a pitch change without preserving duration (SFLength)].

The problem involves the minimum buffer sizes that the audio driver can handle. Some drivers do not properly handle wave buffers that are less than 2,048 or 4,096 bytes in size. As a result, the following explains what happens with SFMix and SFLength on these drivers:

SFMix

When converting from stereo to mono, the output buffer is exactly half the size of the input buffer. If a small buffer is passed in, then an even smaller buffer is output. With low sample rates or a high number of buffers per second, this can cause output buffers to be smaller than 4k (or even 2k).

SFLength

When lengthening data, this plug-in must often use multiple delivery buffers to entirely process an input buffer. The last buffer used will sometimes be very small. For example, if you get a 4,096 byte buffer as input and you lengthen by roughly 101 %, then two buffers will be output. The first will be completely filled, while the second will contain only approximately 40 bytes.

If you encounter these problems, please notify your audio card manufacturer, and request that they fix their drivers. Also, note that (at least in the SFMix case) decreasing the buffers per second that are processed can eliminate the glitches on some audio cards.

Note: Sound Forge 4.0d (and later) contains previewing code that is much more tolerant of buggy wave drivers. Because of these changes, you may never notice the wave driver bugs that exist in many common audio cards. The really good news is that your customers will have less problems as well.

Debugging Tips

Debugging a DirectX Audio Plug-In can sometimes be a little difficult. Fortunately, Microsoft has provided a rather significant amount of information and helper (tracking) functions that greatly aid debugging. You can save yourself a lot of time by reading the "Debugging with ActiveMovie" topic in the ActiveMovie (now called DirectX Media Streaming Services) SDK Help file.

If you are using Sound Forge as a development and testing platform, the following tips can be very helpful:

1. When using Sound Forge as the executable in a debugger, you will not have debugging information for FORGE32.EXE. However, if you "add" your plug-in to the Additional DLLs list in the VC++ environment, it will make setting break points a lot easier. This list is located on the Debug page of the Project Settings dialog, invoked by selecting the **Build** menu, followed by the **Settings** menu option.
2. Always test Real-Time ***and*** NON-Real-Time previews (in addition to actual processing). Each of these cases uses slightly different buffering, and can reveal subtle bugs. It's very easy to get into the habit of always testing with Real-Time previews—but that does not stress all cases. You should switch back and forth frequently.
3. If you are having difficulty getting a break point set at an appropriate location, try using the DbgBreak() and DebugBreak() functions. These functions will force an exception (int 3 on x86 platforms) that the VC++ debugger can handle. Just-in-time debugging needs to be enabled for this feature (it is enabled by default).
4. Sound Forge installs an exception handler on the processing thread of a DirectX Plug-In. This handler helps improve the stability of Sound Forge when running buggy plug-ins. However, the exception handler can also get in the way of locating bugs. You can disable the exception handler with the following registry key:

   HKEY_LOCAL_MACHINE\SOFTWARE\Sonic Foundry\Sound Forge\4.5\ActiveX\
   {your-plug-in-class-id}\Metrics\516 = 0 (default) or 1 (disable handler)

5. Always test with Microsoft's Graph Editor application before shipping a plug-in. Sound Forge and the Graph Editor have very different stress factors, and the combination of the two can help find many bugs.
6. Sound Forge disables alignment faults on RISC processors (this does not apply to x86 machines). This improves stability when running buggy plug-ins. You can enable alignment fault exceptions by setting NoAlignmentFaultExcept equal to 0 in the [Sound Forge] section of the FORGE32.INI file. This is strongly recommended while developing and testing plug-ins, to ensure that alignment faults are not occurring and slowing down processing. In addition, you will need to set the following registry key on non-checked builds of Windows NT:

   Note: Not all applications (including WinHelp) work well when alignment faults are enabled. Though Sonic Foundry's applications should function without incident.

   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\
   EnableAlignmentFaultExceptions = 0 (default) or 1 (enable faults)

7. By default, Sound Forge does not unload a plug-in after it is used. This helps speed up processing for normal users. However, this can make using Sound Forge as a test bed for plug-ins slow because you have to restart the application every time you make a change to the plug-in. You can force Sound Forge to unload plug-ins after being used by setting the following metric in the FORGE32.INI file [Metrics] section:

   2900=1 ; 0 is default, 1 is always unload

Before You Ship

Below is a simple checklist that you should go through before shipping your plug-ins:

1. Make sure you build a full retail version (set nodebug=1 and nmake -a).
2. Verify the version information (by viewing the Properties from the Windows Explorer).
3. Verify that the self-register (and unregister!) features work.
4. Verify that both Graph Editor and Sound Forge can use the plug-in (both for previews and processing). The Sound Forge demo allows this test in case you do not own a copy.
5. Verify that both valid and invalid formats are handled correctly. That is, if your plug-in only handles stereo data, try giving it a mono input—it should fail gracefully. If your plug-in only handles specific sample rates, test that non-supported rates fail gracefully. And so on…

Writing a Setup Program for Plug-Ins

Included in the PIDK is a folder called SELFREG. This folder contains a file, called SELFREG.C, that you can use to implement a program that calls the self-registration routines of a DirectX Audio Plug-In. If you do not wish to use REGSVR32.EXE in your setup program, and your setup program does not already support Self Registration components, then you can use this supplied code.

Writing a DirectX Plug-In Compatible Application

If you are writing an application that you want to support DirectX Plug-Ins, you can make the application compatible for Plug-Ins in one of two ways. The first (and easiest) option is to use Microsoft's default Graph Manager. The second (but more difficult) option is to write a custom Graph Manager.

If you use Microsoft's default Graph Manager, you will need to implement a custom source and rendering filter to stream the data through a filter graph. Microsoft provides examples of this technique in the DirectX Media Streaming Services (DMSS) Software Development Kit (SDK).

Common Presets Between Host Applications

This section describes a common method of saving and restoring user defined presets between host applications. This method was designed in conjunction with several companies that support DirectX Audio Plug-Ins and should become the standard as companies release updates to their products.

Sonic Foundry just released Sound Forge 4.5 and ACID, both supporting this new method of saving and restoring user defined presets. If you are writing a host application, do the users a favor and consider supporting this method.

In the Registry, the key HKEY_CURRENT_USER\Software\DirectShow\Presets contains entries for the filters that have user defined presets. There are also entries made for all filters that do not support the IStaticFilterPreset so that a default entry - (Untitled) - exits.

The filter has a key associated to it based on the CLSID of the filter. For example:

HKEY_CURRENT_USER\Software\DirectShow\Presets\{52CAB881-EE61-11D0-B2DE-444553540000}

The key contains one or more of the following values: String Type or Binary Type

The name of the value is the user defined preset name - i.e. "My really cool Reverb"

The type of the value is determined by the size of the data that the preset needs to save (this is done because large presets could cause the Registry to grow too large).

- If the size of the preset data is **less than or equal to 256 bytes**, the preset data is stored directly in the registry as a Binary Data Type.

- If the preset data size is **greater than 256 bytes**, a file on local storage is created that has an .dxp extension and the value is a String Data Type that contains the full path to the preset data. The name is created on the fly and is determined by the application (typically based on the user preset name). The path is usually determined by the application installation location (doesn't have to be). Since the value of the string is *always* a fully qualified path, the location of the preset data is not critical. Preset file names follow the legal character set for file names. If an application bases the filename on the preset name, it should parse the name and replace illegal characters with legal characters (for example, replace illegal characters with an underline).

An example:

```
HKEY_CURRENT_USER
  Software
        DirectShow
              Presets
                    {52CAB881-EE61-11D0-B2DE-444553540000}
                          My Super Cool Reverb          a0 00 00 00 e3 .....
                          In a HUGE metal Tank                  a0 00 00 00 3e .....
                    {509DEB80-73A1-11D0-AD41-00AA001F6A58}
                          Shure SM58 in Bathroom        d:\Program Files\Sound Forge\SFP00041.DXP
                          Studio Ambiance               d:\Program Files\Sound Forge\SFP00344.DXP
                    {F7E670C0-DB43-11CF-BBFA-00AA00A8658E}
                          (Untitled)                    30 00 30 00 30 00 ...
                          CWPA Favorite Reverb          30 00 30 12 30 00 ....
```

NOTE: The (Untitled) entry is for plug-ins that do not support the IStaticFilterPreset interface.

Known Bugs and Limitations of DirectShow

The DirectX Media 5.2a Software Development Kit (SDK) and the initial release of DirectX Media Stream Services (DMSS, formally known as ActiveMovie 1.0) has a few bugs and limitations that directly affect audio-related plug-ins.

Graph Editor Property Page Dialogs

The Graph Editor application that ships with the DMSS 1.0 and 5.2a SDK does not size the property page wrapper around a plug-in's property pages. The result is that only a portion of large property pages are visible in the Graph Editor. This is a Graph Editor bug.

Gargle Bugs and How to Fix Them

The Gargle example shipped with Microsoft's DMSS 1.0 and 5.2a SDK contains several bugs that, among other things, degrade performance and usability. Microsoft knows about these bugs, and has fixed the first bug for the DirectX Media 5.2a SDK. However, the second bug is still an issue. This section describes each of the known Gargle bugs and how to fix them.

GetSize vs. GetActualDataLength

In the GARGLE.CPP file, the CGargle::Transform function incorrectly calls GetSize rather than GetActualDataLength to determine how much data should be processed. The GetSize function returns the total allocated size of the buffer (which may not be completely filled). The GetActualDataLength function returns the total amount of valid data that is contained in the buffer.

This is a very serious problem if an upstream filter is passing partially-full buffers. If you duplicate this bug, your filter may process parts of the buffer that contain garbage. At best, you are wasting processing time. At worst, you return processed garbage on the other side.

To fix this bug, locate the following line in CGargle::Transform of the GARGLE.CPP file:

```
iSize = pSample->GetSize();
```

Replace the above line of code with:

```
iSize = pSample->GetActualDataLength();
```

OnConnect Memory Leak and Property Updates

In the GARGPROP.CPP file, the CGargleProperties::OnConnect function has two bugs. Below, we describe the bugs and then list replacement code:

1. If OnConnect is called and m_pGargle is non-NULL, Gargle will *assert*, but it will not release the old m_pGargle. This will cause a memory leak, because m_pGargle was never Release()'d. The fix is labeled BUG FIX ONE in the code listed below.

2. If Gargle is already displaying a property page, the page's controls are not updated to reflect the internal state of the new m_pGargle. In fact, Gargle *never* queries the state of m_pGargle to update its controls. This is a serious bug, which, thankfully, is so obvious that it tends to get fixed by everyone. The fix is labeled BUG FIX TWO in the code listed below.

```
// The code listed below is a replacement for the same function in the
// GARGPROP.CPP file of the Gargle example contained in Microsoft's
// ActiveMovie SDK 1.0. This new code fixes two bugs labeled BUG FIX ONE
// and BUG FIX TWO.
//
// These bug fixes will be included in the next release of the ActiveMovie
// SDK.

// OnConnect
//
// Override CBasePropertyPage method.
// Get the interface to the filter.
// init the dialog controls to reflect the current state of the filter
//
HRESULT CGargleProperties::OnConnect(IUnknown * punk)
{
    // Get IGargle interface
    //
    if (punk == NULL)
    {
        DbgBreak("You can't call me with a NULL pointer!!");
        return E_POINTER;
    }

    // OnConnect can be called to update connected object even if
    // the property page is already connected. (This could be optimized
    // to be more efficient if we are given the same object that we
    // already have an interface for.  In that case we would just
    // get object parameters and update our UI to reflect the object state).
    //
    if (NULL != m_pGargle)
    {
        m_pGargle->Release(); // BUG FIX ONE
        m_pGargle = NULL;
    }

    HRESULT hr = punk->QueryInterface(IID_IGargle, (void **) &m_pGargle);
    if (FAILED(hr))
    {
        DbgBreak("Can't get IGargle interface.");
        return E_NOINTERFACE;
    }

    // get current filter properties
    //
    ASSERT(m_pGargle);
    m_pGargle->get_GargleRate(&m_iGargleRate);
    m_pGargle->get_GargleShape(&m_iGargleShape);

    // if we already have a window up, we need to update the controls
    // to match params for the new object we have just connected to.
    //
    if (NULL != m_Dlg)  // BUG FIX TWO
    {
        if (m_hwndSlider)
        {
            int iPos = ConvertToPosition(m_iGargleRate);
            SendMessage(m_hwndSlider, TBM_SETPOS, TRUE, iPos);
        }

        // Set the button text to match.
        SetButtonText(m_Dlg, m_iGargleShape);
    }

    return NOERROR;
}
```

Sound Forge Issues

The following lists describe what we consider to be significant limitations in Sound Forge when supporting DirectX Audio Plug-Ins.

General Limitations

1. Format converters (other than mono to/from stereo) are not supported. Therefore, a resample or bit-depth converter can not be written for use in Sound Forge. Please note that a requantizer (like Waves L1) can be written if the bits per sample does not change. Rather, you simply need to requantize the data in-place.
2. Source and Renderer plug-ins are not supported. Therefore, you cannot currently write "generators" (synthesizers) or "scopes" for use in Sound Forge.
3. Only 8- and 16-bit PCM and 32-bit float connections are supported.
4. The allocator does not allow a plug-in to specify a larger output buffer than the input (this is entirely valid). Properly written plug-ins will use GetDeliveryBuffer in a loop to accommodate this limitation. However, this limitation also makes some plug-ins more difficult to write.

Sound Forge 4.0 Limitations with the DirectX 5.2a SDK

The DirectX 5.2a SDK supports a slightly different plug-in registration process than the ActiveMovie 1.0 SDK. Unfortunately, this new registration location is not compatible with Sound Forge 4.0d, build 200 and earlier.

An update to Sound Forge 4.0d (later than build 200) can be obtained from Sonic Foundry's web site.

Additional Sound Forge Issues

The following list describes additional issues that you should understand:

Sound Forge requires at least one TRANSFORM_PROPS initializer. When defining the PRESET_INITIALIZER, the first is always assumed to be the "default" or "untitled" preset. It is used when no other preset is specified. By design, Sound Forge will display "untitled" for the first preset.