Echo Generic Driver

Release A.0

Echo uses a cross-platform set of C++ classes as the base for our drivers; these classes encapsulate the hardware in an OS-independent manner. Collectively, this set of classes is known as the "generic driver." The generic code has been used to develop drivers for Windows NT 4.0, Mac OS, and WDM.

All of the hardware supported by the generic driver follows the same basic design. There is a PCI card with a Motorola DSP chip. The Motorola DSP has a built-in PCI interface, so communication to and from the PCI card is handled by the DSP. Almost all communication and data transport for the card is done via bus mastering.

Being familiar with the products is essential. If you want to develop a driver for this hardware, you are strongly encouraged to play with the existing Windows or Mac driver so that you understand concepts such as the digital mode switch. At the very least, you should grab the product manuals from our web site (www.echoaudio.com) and read them.

Open source license

This source code is provided under the terms of the following license agreement. If you do not agree to the terms of this license, you may not make use of this source code.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal with the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

- Neither the name of Echo Digital Audio, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

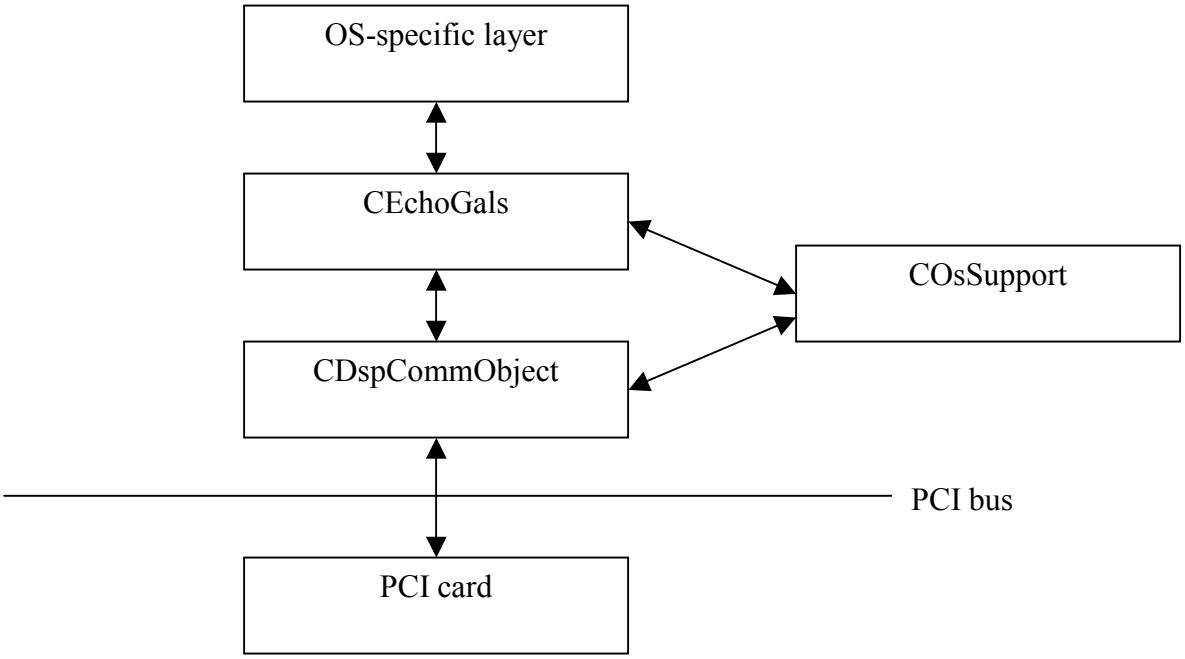
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.

The language for this license agreement was taken from the University of Illinois/NCSA Open Source License.

Layout

Here's the basic layout of a driver using the generic code:



The OS-specific layer just that – it is not part of the generic code. This part needs to be newly developed for every platform.

The CEchoGals class is the upper edge of the generic driver – it is the interface between the OS-specific code and the generic driver.

CDspCommObject is the lower edge – it handles communication between the generic driver and the PCI card.

COsSupport is a utility class used to do things like allocate memory. COsSupport must be rewritten for every platform.

Each Echo hardware product derives its own CEchoGals and CDspCommObject class: for example, Layla24 has CLayla24 (derived from CEchoGals) and CLayla24DspCommObject (derived from CDspCommObject). You don't create objects of CEchoGals and CDspCommObject; instead, you look at the PCI subsystem ID and create the appropriate CEchoGals-derived object for that card. The CEchoGals-derived object, in turn, creates the appropriate CDspCommObject-derived object.

Concepts

Here are some terms used by the generic driver:

Pipe: A pipe moves audio data across the PCI bus. If you want to play some audio, you will open an output pipe and use that. Pipes are numbered starting at zero.

Bus: A bus is a physical connection to the real world, such as the ¼" connector on the breakout box. Busses are also numbered starting at zero.

Most cards have the same number of pipes and busses – each pipe is mapped one-to-one to a bus. The connection between pipes and busses cannot be changed.

Virtual outputs: One product (Mia) has support for virtual outputs – that is, Mia has more output pipes than output busses. The data from the output pipes is mixed together and directed to the output busses as specified by the driver. Future products will probably support virtual outputs as well.

To determine if a card has virtual outputs, call CEchoGals::GetCapabilities and look at the fHasVmixer flag.

Pipe index: A pipe index is used to either refer to an input or output pipe. Input and output pipes are given unique indices; analog output pipes are numbered first, followed by digital output pipes, followed by analog input pipes, followed by digital input pipes.

For example, take Gina24. Gina24 has 8 analog outputs, 8 digital outputs, 2 analog inputs, and 8 digital inputs. Since Gina24 does not support virtual outputs, each pipe is tied directly to a bus. So:

Bus	Pipe index
Analog out 0	0
Analog out 1	1
Analog out 2	2
Analog out 3	3
Analog out 4	4
Analog out 5	5
Analog out 6	6
Analog out 7	7
Digital out 0	8
Digital out 1	9
Digital out 2	10
Digital out 3	11
Digital out 4	12
Digital out 5	13
Digital out 6	14
Digital out 7	15
Analog in 0	16
Analog in 1	17
Digital in 0	18
Digital in 1	19
Digital in 2	20
Digital in 3	21
Digital in 4	22
Digital in 5	23
Digital in 6	24
Digital in 7	25

A number of functions in the generic driver take a pipe index as a parameter. To make a pipe index, use the handy CEchoGals::MakePipeIndex function.

Family: A group of products. To make the tester's lives easier, we split the products up into families. Families break down like this:

Echogals: Darla20, Gina20, Layla20, and Darla24
Echo24: Gina24, Layla24, Mona, and Mia

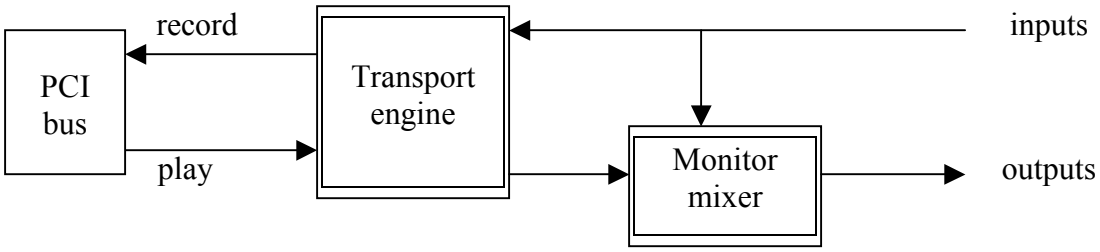
Historically, we have had one driver binary for Echogals and another for Echo24, both built from the same source code. More card families will be added in the future; the wise driver developer will bear this in mind.

The generic code has a set of #defines that build the code differently, depending on the family. These are laid out in family.h. To build a driver for the different families, you will need to set the appropriate #define for the family.

For Echogals, #define ECHOGALS_FAMILY.
For Echo24, #define ECHO24_FAMILY.

Hardware block diagram

Here's a block diagram of what goes on inside the DSP:



The arrows labeled "record" and "play" are the pipes – data moving over the PCI bus.

The inputs and outputs on the right-hand side of the diagram are the busses – real, physical connections to the outside, real, physical world.

The transport engine does two things:

- Converts data between the DSP's internal 24 bit fixed point data format and whatever format has been specified by the driver

- Transports the data over the PCI bus

The monitor mixer lets you listen to the inputs; it can potentially mix every input to every output simultaneously.

20-bit hardware versus 24-bit hardware

Some of the products supported by this code have 20-bit analog converters – specifically, Darla20, Gina20, and Layla20. All of the other products have 24-bit converters.

Despite this, there really isn't any difference in how the different products handle audio data. All of the products have a 24-bit internal signal path – that is, the DSP treats all audio data as 24 bit; 20-bit data is simply padded with four zero bits at the bottom.

The upshot is that you, as a driver developer, don't need to worry about the difference.

Constructing the generic objects

Start out by determining which card you have. You do this by looking at the PCI configuration space.

Note that there are other PCI cards in the world that use the 56301 and 56361; make sure you verify the PCI vendor ID, device ID, subvendor ID, and subsystem ID.

All the cards have the same vendor ID:

Vendor ID:	0x1057	Motorola

There are two possible device IDs:

Device ID:	0x1801	56301 DSP
	0x3410	56361 DSP

Darla20, Gina20, Layla20, Darla24, some Gina24, and some Mona cards use the 56301 DSP.

Layla24, Mia, some Gina24, and some Mona cards use the 56361.

All the cards have the same subvendor ID. Checking the subvendor ID is very important; there are other cards out there that use these same DSP chips:

Subvendor ID:	0xECC0	(Get it?)

The particular product you have is determined by looking at the subsystem ID:

Subsystem ID:	0x0010	Darla20
	0x0020	Gina20
	0x0030, 0x0031	Layla20
	0x0040, 0x0041	Darla24
	0x0050, 0x0051	Gina24
	0x0060	Layla24
	0x0070, 0x0071, 0x0072	Mona
	0x0080	Mia

The upper 12 bits of the subsystem ID tell you which product it is, while the lower four are used for the hardware revision. Generally, drivers mask off the upper 12 bits and use that to determine which object to construct.

Now, you'll need to rewrite COsSupport. The driver includes COsSupportWDM; you'll need to take this and rewrite it as COsSupportOSX or COsSupportTimexSinclair or whatever. Depending on the requirements of your operating system, you may or may not need to instantiate a separate OS support object for each PCI card.

You'll also need to rewrite ossupport.h and use the appropriate preprocessor #defines to make sure that your new class is used when you build the generic code.

So, now that you've written your OS support class, have your driver create an OS support object. Then, create an appropriate CEchoGals-derived object, based on the subsystem ID.

```
      ntStatus = WriteConfigSpace(DeviceObject,&config);
      if (!NT_SUCCESS(ntStatus))
         return ntStatus;
   }

   m_wCardType = config.u.type0.SubSystemID & 0xfff0;
   m_wCardRev =  config.u.type0.SubSystemID & 0x000f;

   //------------------------------------------------------------------
   //
   // Create the generic driver objects
   //
   //------------------------------------------------------------------

   //
   // Make the OS support object
   //
   m_pOSS = new COsSupport(config.DeviceID);
   if (NULL == m_pOSS)
      return STATUS_INSUFFICIENT_RESOURCES;

   //
   // Make the EchoGals object
   //
#ifdef ECHOGALS_FAMILY

   switch (m_wCardType)
   {
      case DARLA :
         m_pEG = new CDarla(m_pOSS);
         break;

      case GINA :
         m_pEG = new CGina(m_pOSS);
         break;

      case LAYLA :
         m_pEG = new CLayla(m_pOSS);
         break;

      case DARLA24 :
         m_pEG = new CDarla24(m_pOSS);
         break;

      default :
         DbgPrint("Card type 0x%x not supported by this”
                  “ driver!\n",m_wCardType);
         return STATUS_UNSUCCESSFUL;
   }

#endif // ECHOGALS_FAMILY


#ifdef ECHO24_FAMILY

   switch (m_wCardType)
   {
      case GINA24 :
         m_pEG = new CGina24(m_pOSS);
         break;

      case LAYLA24 :
         m_pEG = new CLayla24(m_pOSS);
         break;

      case MONA :
         m_pEG = new CMona(m_pOSS);
         break;

      case MIA :
         m_pEG = new CMia(m_pOSS);
```

```
         break;

      default :
         DbgPrint("Card type 0x%x not supported by this”
                       “ driver!\n",m_wCardType);
         return STATUS_UNSUCCESSFUL;
   }
#endif // ECHO24_FAMILY

   if (NULL == m_pEG)
   {
      return STATUS_INSUFFICIENT_RESOURCES;
   }

   //---------------------------------------------------------------------------
   //
   // Find and init the hardware
   //
   //---------------------------------------------------------------------------

   //
   // Get plug-n-play info
   //
   PCM_PARTIAL_RESOURCE_DESCRIPTOR  resource;
   DWORD                            dwIrq;
   PHYSICAL_ADDRESS                 PhysAddr;
   PVOID                            pDspRegs;

   resource = ResourceList->FindUntranslatedInterrupt(0);
   dwIrq = resource->u.Interrupt.Level;

   resource = ResourceList->FindTranslatedMemory(0);
   PhysAddr = resource->u.Memory.Start;

   DOUT(DBG_PRINT,("Physical address is 0x%x",PhysAddr.LowPart));

   //
   // PhysAddr is just what you think – it’s a physical address.
   // You can’t dereference it to access the hardware registers; calling
   // MmMapIoSpace will convert it to a logical pointer.
   //
   m_pDspRegs = (PDWORD) MmMapIoSpace(PhysAddr,DSP_REG_WINDOW,MmNonCached);
   if (NULL == m_pDspRegs)
   {
      return STATUS_INSUFFICIENT_RESOURCES;
   }

   //
   // Pass the logical pointer to the memory-mapped DSP registers to the
   // generic code.
   //
   m_pEG->AssignResources(m_pDspRegs,szNoName);

   //
   // Now that AssignResources has been called, call InitHw to try and load
   // the DSP.
   //
   Status = m_pEG->InitHw();

   if (ECHOSTATUS_OK == Status)
   {
      //
      // Everything is wonderful.
      //
   }
   else
   {
      //
      // Oh, crap!  We’re all going to die!
      //
   }
```

Querying the generic driver

Once you have a pointer to a CEchoGals-derived object, you can start to build a driver around it. Our approach is to avoid having conditional code in the OS-specific layer that depends on the card type; generally, there is just the one switch statement (as demonstrated above) for building the generic object in the first place.

Once that is done, we call CEchoGals::GetCapabilities. This fills out an ECHOGALS_CAPS structure that has essentially all of the information that you need for this card. The ECHOGALS_CAPS structure is generally sufficient for making decisions about how many inputs and outputs to expose to the OS and so forth.

For a breakdown of the ECHOGALS_CAPS structure, please refer to EchoGalsXface.h

Some hardware capabilities are exposed via the driver flags interface; these are read via CEchoGals::GetFlags. Some flag bits may be set or cleared, such as ECHOGALS_FLAG_SYNCH_WAVE, which, if set, causes the generic code to attempt to synchronize starting transport over multiple pipes.

Other flag bits are read only – these are denoted by the word "ROFLAG" in their name. For example, if ECHOGALS_ROFLAG_SUPER_INTERLEAVE_OK is set, then you know that this card can handle super interleave (defined below). Since these flags are read-only, you cannot clear them – they are there for informational purposes only.

COsSupport

COsSupportWDM.cpp is provided as an example. Most of what happens in here has to do with kernel memory allocation.

OsAllocateNonPaged is used to (surprise!) allocate non-paged memory. Non-paged memory is defined as memory that is always physically resident – that is, the virtual memory manager won't swap this memory to disk. Nonpaged memory can be accessed from the interrupt handler.

OsAllocateNonPaged is not a member of COsSupport since you need it to construct the COsSupport object; it's used in the overridden new handler.

Nonpaged memory has no particular requirements on address alignment.

COsSupport::OsPageAllocate is used for allocating memory that is read and written by the DSP. This means that any memory allocated by OsPageAllocate must be nonpaged and should be aligned on at least a 32 byte address boundary. It must be aligned on at least a 4 byte boundary.

Audio transport interface – data formats and sample rates

Audio transport refers to either playing back or recording audio. Transport is done via the bus-mastering interface on the DSP chip; the driver's job is to set up various data structures in memory and set up the DSP appropriately.

To find out if the hardware can handle a given sample rate, call CEchoGals::QueryAudioSampleRate. Supported sample rates vary from card to card.

All of the cards support the following playback data formats:

Playback formats

8 bit unsigned mono → stereo
8 bit unsigned stereo → stereo
16 bit signed little-endian mono → stereo
16 bit signed little-endian stereo → stereo
32 bit signed little-endian mono → mono
32 bit signed little-endian mono → stereo
32 bit signed little-endian stereo → stereo
32 bit signed big-endian mono → mono

The notation "mono → mono" means that data that goes into a single pipe comes out of a single bus; that is, if you start playback on pipe 2 and select "32 bit signed mono → mono", the audio will come out of output bus 2 and only output bus 2. This format is used for ASIO support under Windows.

Mono → stereo formats work just as you'd expect; start playback on pipe 2 and you hear audio from output busses 2 and 3. The same signal is sent to both output busses.

Stereo data is interleaved left-right in memory.

For the 32 bit formats, only the upper 24 bits are significant; the lower 8 bits are ignored by the DSP. This format is also commonly referred to as 32 bit *left-justified*.

Record formats

Mono → 8 bit unsigned mono with dithering
Stereo → 8 bit unsigned stereo with dithering
Mono → 16 bit signed little-endian mono with dithering
Stereo → 16 bit signed little-endian stereo with dithering
Mono → 16 bit signed little-endian mono without dithering
Stereo → 16 bit signed little-endian stereo without dithering
Mono → 32 bit signed little-endian mono
Stereo → 32 bit signed little-endian stereo
Mono → 32 bit signed big-endian mono

These work similarly to the playback formats; the main difference is whether the DSP dithers the audio. Typically, the "16 bit with dithering" format is used for the analog inputs, whereas the "without dithering" format is used for the digital inputs.

Super interleave

Some cards also support interleaving by more than two; in the generic code, this is known as "super interleave". To find out if a particular card supports super interleave, call CEchoGals::GetFlags and see if ECHOGALS_ROFLAG_SUPER_INTERLEAVE_OK is set.

If it is set, then the card can interleave by n, where n is either one or an even number. You cannot interleave past the number of pipes on the card; for instance, Layla24 has 16 output pipes. If you start playback on pipe 6, you can interleave by 1, 2, 4, 6, 8, or 10. Interleave of 12, 14, or 16 is not allowed. If you start on pipe 0, you can interleave by up to 16.

Super interleave is contiguous in terms of the pipe numbers; say that you start on pipe 6 and specify an interleave of 4. The data will come out of output busses 6 through 9.

For now, super interleave data is always 32 bit left-justified little-endian. Support for big-endian super interleave may be added in the future.

Super interleave works for both input and output pipes.

The comm page

The comm page isn't necessarily something you need to understand – hopefully, you will be able to treat the generic code as a black box that just magically works. If you like, skip to the next bit.

If you do want to understand it, the comm page is one of the fundamental mechanisms for communication between the DSP and the driver. Specifically, the comm page is a page of memory that is shared by the DSP and the driver - the term "comm page" means that the page is used for *communication* and is used in *common* by the driver and the DSP simultaneously.

The actual page of memory used for the comm page is allocated by the driver as fixed memory – that is, the page is resident in memory at all times, since it is accessed via bus mastering by the DSP. Since this was originally developed for Windows 95 on a Pentium, the comm page is designed to fit within a single 4096-byte page; the generic code uses the COsSupport::OsPageAllocate to create a 4096-byte block on a 4096-byte aligned address. The physical address of the comm page is passed to the DSP as part of the DSP loading routine.

The definition of the comm page structure can be found in CDspCommObject.h. The comm page itself is part of the CDspCommObject class.

The comm page has a number of different applications; for example, if the DSP has metering turned on, it is constantly writing peak and VU meter values back to the comm page. The comm page is also used to send parameters to the DSP; for example, volumes, physical addresses of scatter-gather lists, and so forth.

Audio transport interface – generic driver functions

To actually play or record, you need to do the following:

Call CEchoGals::OpenAudio. This reserves a pipe for your exclusive use. If the pipe is already open, the call will fail.

Note that if you specify an interleave factor of more than one, you will not only reserve that pipe but all of the pipes covered by the interleave factor. For instance, if you specify an interleave of two, you will reserve the specified pipe and the one after it.

Once you have opened a pipe, you need to set the data format. Call CEchoGals::QueryAudioFormat to verify that the card can handle a given audio format; call CEchoGals::SetAudioFormat to, well, set the audio format.

Set the sample rate by calling CEchoGals::SetSampleRate.

Now that you have the pipe open and the format set, you need to start building the scatter-gather list. Scatter-gather lists are managed by the CDaffyDuck class (the meaning of the name is an obscure, pointless inside joke).

A CDaffyDuck object is created automatically when you call OpenAudio. To get the pointer to the CDaffyDuck object for a pipe, call CEchoGals::GetDaffyDuck. (Note that you don't have to clean up the CdaffyDuck object; this is done for you when you call CloseAudio).

You will need to fill the daffy duck with pointers to audio data; see below for more information on that topic.

Once you have filled the daffy duck, you can call CEchoGals::Start. There are several variations on this method that let you start a single pipe, a group of pipes, or all the pipes that are already open. Calling start causes the DSP to begin audio transport according to the scatter-gather list.

The DSP will generate interrupts back to the driver; you will need to call functions within the generic driver to clear the hardware interrupt and service MIDI input; see below for the section on interrupt handling.

The DSP will continue to traverse the scatter-gather list, transporting audio, until it runs out of scatter-gather list entries.

Calling CEchoGals::Stop tells the DSP to pause audio transport; it stops transporting, but preserves the current internal state of the pipe. Calling Start again will resume transport from where you left off.

Calling CEchoGals::Reset stops and resets this pipe; the internal state of the pipe is reset within the DSP.

Scatter-gather lists

Concept

All PCM audio data is moved to and from the DSP via PCI bus mastering. The DSP must therefore be informed as to where in memory to play from or record to. This is handled by a scatter-gather list, or, more whimsically, a CDaffyDuck object.

Bear in mind that you don't necessarily need to understand this, unless the OS for which you are developing has substantially different requirements for how audio data is represented in memory. This explanation is here to make the generic code less of a black box.

Setting up the scatter-gather list correctly is very important; doing it wrong can easily crash your machine. If you don't know what a scatter-gather list is or don't understand physical versus logical memory, you'll need to look elsewhere; that's beyond the scope of this document.

CDaffyDuck objects are constructed automatically when you open a pipe. You can access the duck for a pipe by calling CEchoGals::GetDaffyDuck.

When you start transport, the DSP looks for the starting address of the scatter-gather list in the comm page (to see how this is set up, check out CDspCommObject:: SetAudioDuckListPhys). Since the DSP will be asserting this address on the PCI bus, this is a *physical* address. Once the DSP has the starting address, it begins to parse the SG list.

SG list entries are 8 bytes – first, four bytes for the physical address and then four bytes for the byte count. The DSP has two modes for interpreting SG lists – cyclic and non-cyclic. Cyclic mode means that this SG list is for a circular buffer, such as is used by ASIO, DirectSound, or OS X. Non-cyclic mode means that the SG list represents a non-circular chain of buffers, such as is used by Windows wave drivers or WDM drivers.

For non-cyclic mode, the DSP keeps a count of how many buffers there are in memory. When the DSP runs out of memory, it stops reading the SG list until the buffer count goes up again. The count is incremented every time that CDspCommObject::AddBuffer is called (you wouldn't call that directly; it gets called by the CDaffyDuck object) and is decremented by the DSP as it processes the SG list.

For cyclic mode, the buffer count is ignored.

Here is a pseudocode rendering of how the DSP processes the non-cyclic SG list entries:

```
while (transport is active)
{
   if (current SG byte count is non-zero)
   {
      transport a block of audio
      decrease the SG byte count
      continue;
   }

   if (buffer count != 0)
   {
      get new SG list entry – fetch 8 bytes at SG list pointer
      SG list pointer += 8
      Store SG byte count and SG phys addr

      if (0 == SG phys addr && 0 == SG byte count)
      {
         // Double-zero means “end of buffer”
         generate an IRQ
         decrement the buffer count for this pipe
         continue;
      }
      else if (0 != SG phys addr && 0 == SG byte count)
      {
         SG list pointer = SG phys addr
         continue;
      }
   }

}
```

So, most of the time, an SG list entry points to an audio buffer. If the count is zero and the address is non-zero, this list entry is a pointer to another list entry. If both are zero, this is the end of a buffer; the DSP generates an interrupt back to the host, decrements the buffer count, and goes again.

Cyclic mode is somewhat different, but not much:

```
while (transport is active)
{
   if (current SG byte count is non-zero)
   {
      transport a block of audio
      decrease the SG byte count
      continue;
   }

   get SG list entry
   SG list pointer += 8
   Store SG byte count and SG phys addr

   if (0 == SG phys addr && 0 == SG byte count)
   {
      // Double-zero means “end of buffer”
      generate an IRQ
      continue;
   }
   else if (0 != SG phys addr && 0 == SG byte count)
   {
      SG list pointer = SG phys addr
      continue;
   }

}
```

An SG list must therefore be set up with the correct sequence of 8-byte SG list entries.

Scatter-gather lists - Implementation and usage

CDaffyDuck implements the scatter-gather list as a circular buffer of list entries. Please be aware that this circular buffer of SG list entries does not necessarily point to a circular audio buffer; both cyclic and non-cyclic SG lists are implemented using the same circular buffer of list entries.

Note that the code for CDaffyDuck uses the term "mapping" fairly often; a mapping is a physical block of memory that is part of an audio buffer. CDaffyDuck actually keeps two parallel arrays –the SG list itself and a second array for the mappings. When you want to add an entry to the SG list, you actually add a mapping to the CDaffyDuck object; this, in turn, adds a new entry to both arrays.

The idea here is that typically, you will need to translate the logical address of an audio buffer to one or many physical addresses. This is done by the OS-specific layer. Once you have done that translation and want to make this buffer accessible to the DSP, you call CDaffyDuck::AddMapping one or more times, depending on the number of physically contiguous blocks you want to pass to the DSP.

Here's an example of how to build a circular double buffer- the DSP will interrupt you halfway through the buffer and again at the end. This example is taken from the Mac OS9 driver:

```
//
// Make a circular double buffer for this pipe
//
CDaffyDuck *pDuck;

pDuck = m_pEchoGals->GetDaffyDuck(wPipeIndex);
if (NULL == pDuck)
{
   ECHO_DEBUGPRINTF(("\tCould not get daffy duck pointer!\n"));
   return ECHOSTATUS_INVALID_CHANNEL;
}

// first half...
pDuck->AddMapping(dwPhysAddr,
                  dwHalfBufferSizeBytes,
                  0,    // the tag doesn't matter for a circular buffer
                  TRUE, // yes, generate an interrupt after this half-buffer is done
                  dwNumFreeEntries);

// second half
dwPhysAddr += dwHalfBufferSizeBytes;
pDuck->AddMapping(dwPhysAddr,
                  dwHalfBufferSizeBytes,
                  0,    // again, the tag doesn't matter
                  TRUE, // yes, generate an interrupt after this half-buffer is done
                  dwNumFreeEntries);

// call Wrap to turn this into a circular buffer
pDuck->Wrap();
```

Again, for this to work properly, you would need to open the pipe in cyclic mode.

Non-cyclic mode is somewhat more complex; in cyclic mode you set it up once and you're done. In non-cyclic mode, you are constantly adding mappings to the duck and

constantly calling CDaffyDuck::ReleaseUsedMapping. Calling ReleaseUsedMapping checks to see if the DSP has consumed the oldest mapping in the list; it does this by looking at the 64-bit DMA position of the DSP and comparing it to the 64-bit DMA position of the end of this mapping. If the DSP position is greater than the end position, the DSP is done with this mapping and it may be safely removed from the list.

Here's some code from the WDM driver that adds and removes mappings to the list; typically, you would call this after receiving an interrupt. Note that this code has been simplified in the interests of clarity.

Removing old mappings:

```
//
// Ask the duck about released mappings
//
ULONGLONG   ullDmaPos;
ECHOSTATUS  Status;
DWORD       dwTag;

m_pEG->GetDmaPos(m_wPipeIndex,&ullDmaPos);

do
{
   Status = m_pDuck->ReleaseUsedMapping(ullDmaPos,dwTag);
   if (ECHOSTATUS_OK != Status)
      break;

   //
   // Release this mapping to port class
   //
   PortStream->ReleaseMapping( (PVOID) dwTag);

} while (1);
```

Note that the 64-bit DMA position is first obtained from the generic driver and then passed to the duck.

To add new mappings:

```
//
// Get a bunch of mappings; must be at least two free entries
// in the daffy duck for this channel
//
dwNumFreeEntries = m_pDuck->GetNumFreeEntries();
Status = ECHOSTATUS_OK;

while ((dwNumFreeEntries >= 2) && (ECHOSTATUS_OK == Status))
{
   ntStatus = PortStream->GetMapping(  (PVOID) m_dwNextTag,
                                       &PhysAddr,
                                       &VirtAddr,
                                       &dwByteCount,
                                       &dwFlags );

   if ( !NT_SUCCESS(ntStatus))
   {
      // If GetMapping fails, it's OK – it just means that there's no
      // more audio data right now.
      break;
   }

   Status = m_pDuck->AddMapping( PhysAddr.LowPart,
```

```
                                    dwByteCount,
                                    m_dwNextTag,
                                    dwFlags,
                                    dwNumFreeEntries);   // dwNumFreeEntries is passed
                                                         // by reference

      m_dwNextTag++;
   }

   //
   // Put a double zero at the end of these mappings if port class didn't
   //
   if (NT_SUCCESS(ntStatus) && (0 == dwFlags))
      m_pDuck->AddDoubleZero();
```

Two things to notice here – one is that AddMapping should only be called if there are at least two free entries in the scatter-gather list. If you call AddMapping with dwInterrupt set to 1, then this consumes two entries in the list – one for the audio data pointer and one for the double-zero.

The other useful thing to notice is the call to AddDoubleZero – sometimes, you need to add a double zero to the end of the list so the DSP will generate an audio interrupt and keep things flowing. Hence, the call to AddDoubleZero after the loop.

One final note – the scatter-gather list is obviously read by the DSP; it must therefore be written to memory in little-endian format. AddMapping does this for you.

Handling interrupts

Hardware interrupts are obviously very platform-specific. Determining which interrupt to use, hooking an interrupt handler, etc – these are not covered by the generic driver.

However, once you have your interrupt handler hooked up, dealing with them is straightforward. The generic code provides a single routine for dealing with interrupts:

```
virtual ECHOSTATUS CEchoGals::ServiceIrq(BOOL &fMidiReceived);
```

ServiceIrq should be called from within your hardware interrupt hander. It does the following:

-Checks to see if this card is generating the interrupt; if not, it returns the value ECHOSTATUS_IRQ_NOT_OURS. This is necessary to support interrupt sharing.

-Looks for MIDI input data in the comm page; the DSP will write new MIDI input data to the comm page before generating the IRQ. If new MIDI input data has been written, ServiceIrq calls the current instance of CMidiInQ to store the MIDI input data.

-Clears the hardware interrupt

If new MIDI data was received, then the fMidiReceived flag will be set to TRUE.

Mixer interface

The mixer interface allows you to set volumes, clock settings, and so forth. Just about any hardware setting you'd care to make is done through the mixer interface.

To use the mixer interface, you should open it. This is done with CEchoGals::OpenMixer (to clean up when you're done, call CEchoGals::CloseMixer).

```
//
// Open and close the mixer
//
ECHOSTATUS OpenMixer(DWORD &dwCookie);
ECHOSTATUS CloseMixer(DWORD dwCookie);
```

The value dwCookie is used to uniquely identify each mixer client; this value is assigned by the generic driver. Pass this same cookie value back when you call CloseMixer so that the right client gets cleaned up.

There are two calls for getting and setting mixer controls:

```
//
// Process mixer functions
//
virtual ECHOSTATUS CEchoGals::ProcessMixerFunction
(
   PMIXER_FUNCTION   pMixerFunction,   // Request from mixer
   INT32 &           iRtnDataSz        // # bytes returned (if any)
);

//
// Process multiple mixer functions
//
virtual ECHOSTATUS CEchoGals::ProcessMixerMultiFunction
(
   PMIXER_MULTI_FUNCTION   pMixerFunctions,  // Request from mixer
   INT32 &                 iRtnDataSz        // # bytes returned (if any)
);
```

As you may have gathered from the names, they both handle mixer functions. ProcessMixerFunction handles a single mixer function; ProcessMixerMultiFunction handles more than one.

For the definition of MIXER_FUNCTION and MIXER_MULTI_FUNCTION, please refer to MixerXface.h.

You can also find out what mixer controls have been changed by other mixer clients; this is where the cookies come in.

```
//
// Get all the control notifies since last time this was called
// for this client.
//
ECHOSTATUS CEchoGals::GetControlChanges
(
   PMIXER_MULTI_NOTIFY  pNotifies
);
```

This gives you a list of controls that have been changed by all clients. Note that MIXER_MULTI_NOTIFY is a variable length structure; please see MixerXface.h for details.

Miscellaneous

You may need to explicitly set your compiler to build in C++ mode; the generic code uses C++-specific syntax.

Technical support

Support for this source code will be available on a "when the developer isn't too busy to answer questions" basis. Support will only be provided via email – send your questions to devsupport@echoaudio.com.

This is a work in progress; if you see something that needs to be better documented, or have a suggestion, let us know at the same address.

If you're planning on a project that uses this code, drop us a line and let us know – we're interested to see who makes use of all this.