Digidesign Audio Engine Plug-In Software Development Kit

Version 7.3

Digidesign Inc.
2001 Junipero Serra Boulevard
Daly City, CA 94014-3886
tel: 650-731-6300
fax: 650-731-6399

Part I: Introduction

Chapter 1: Getting Started

Greetings Developer! Welcome to Digidesign's Plug-in Software Development Kit! This kit contains information on how to create plug-in software modules for Pro Tools® or any third-party application that is DAE (Digidesign Audio Engine™) aware.

This SDK is maintained by the Developer Services Engineering group at Digidesign. We at Developer Services are here to make your development process as fast, fun, and productive as possible. Our goal is to help you develop world class plug-ins.

What You Need to Know

This documentation assumes that you are familiar with C++ object-oriented programming. C++ concepts such as classes, methods, dispatching calls, inheritance, and overriding are used throughout this documentation. All source code provided in this Plug-in Software Development Kit is written in the C++ programming language. You must be able to write C++ source code that will build off of the Plug-In Library that we have provided.

If you plan to develop a TDM plug-in that will implement a DSP algorithm, you must have knowledge of Motorola 56K DSP assembly language. Plug-ins that do not require a DSP can be developed without any knowledge of 56K assembly.

It is also useful to have at least some basic working knowledge on how to use Pro Tools or any DAE-aware application that supports AudioSuite and/or TDM plug-ins, so that you can better understand what the end-user will expect when he/she uses your plug-in.

The Developer Website

http://developer.digidesign.com

The Developer Website is one-stop-shopping for all your development needs. You should visit frequently in order to stay current with the latest news, builds, and documentation. As a plug-in developer, you will have access to four key development areas: *Home, Tech Notes, Pro Tools*, and *Plug-In SDK*.

Development Environment

Though algorithm development can be totally cross-platform, design of your GUI is unfortunately tied to Macintosh Resources if you are using the Digidesign UI framework. Developer Services is currently looking at various solutions to get around this Macintosh dependency. For the time being, however, you will need access to a Mac in order to build and test your GUI if you use the Digidesign UI framework.

Hardware

If you are planning on developing only AudioSuite and RTAS plug-ins, you will need either an MBox2, MBox, Digi 002, or Digi 002 Rack hardware system (Digi 001 is no longer supported, as of PT 6.7). If you

are planning on building TDM plug-ins, you will need an HD or HD|Accel Hardware System (MIX is no longer supported, as of PT 6.7). To inquire about special deals for developers on Digidesign hardware, contact Ed Gray at Ed_Gray@digidesign.com.

Macintosh

Please consult the "Support"→"Compatibility" section of the Digidesign website (www.digidesign.com) for the latest supported machines from Apple. The README file in each SDK lists the supported development environments for that particular SDK.

Windows

Please consult the "Support"→"Compatibility" section of the Digidesign website (www.digidesign.com) for the latest recommended Windows machines. This is especially important for using Pro Tools on Windows, as certain machines simply DO NOT WORK with Pro Tools. Please shop carefully. The README file in each SDK lists the supported development environments for that particular SDK.

Pro Tools

Pro Tools LE will only run LE hardware, e.g. MBox 2, MBox, Digi 002, or Digi 002 Rack. Pro Tools TDM will only run on TDM hardware. We provide debug builds of Pro Tools to assist you with your development. These debug builds are called "Pro Tools Developer Build" and they are only available to third-party developers. The Developer build enables a few special functions, like a console window for debugging output.

Contacting Developer Services for Support

At various points in your development, you will need support for building and debugging your plug-in. All technical questions should be sent to **devservices@digidesign.com**. This is an email alias for all the members of the Developer Services Engineering group.

Sending an email to devservices@digidesign.com will create a support ticket in our support issue tracking system. These tickets are answered in the order they are received.

Comment [MSOffice1]: This should be removed, no? jk 10/21

Giving Feedback on this SDK

Every effort is made to keep the SDK as accurate and up-to-date as possible. If you have any comments on the SDK or its documentation, or find errors that you'd like to report, we would be happy to hear from you. You can send feedback to `devservices@digidesign.com`

Thanks for your support and development effort!

Chapter 2: The Plug-In Environment

There are currently three types of DAE plug-ins: AudioSuite, Real-Time AudioSuite, and TDM.

■ AudioSuite (AS) plug-ins perform non-realtime file-based processing entirely on the host CPU.

■ Real-Time AudioSuite (RTAS) plug-ins perform real-time non-destructive processing entirely on the host CPU.

■ TDM plug-ins perform real-time non-destructive processing on TDM enabled hardware, using the host for UI support.

Plug-in developers can choose among these platforms to build their plug-ins, as each one of them offers advantages and trade-offs in functionality, power, and development effort.

Plug-In History

The very first plug-in was developed in 1912 by an American German immigrant of the name Wilheim Crowder. Working as a metalsmith apprentice in the suburbs of New York City, he stumbled upon the idea of allowing third-party code fragments to cooperate with his newly devised Wilheim Audio Engine. This stunning breakthrough poised him to be the next American-Dream-rags-to-riches Time cover story. Fortunately, his naviete prevented him from patenting the idea, and we, the ruthless and cunning Digidesign, leveraged the technology away from him.

Modern DAE plug-ins are comprised of shared libraries. As a plug-in developer, you will be writing a support library which adds new functionality to a DAE aware application. These specialized libraries can attach to a DAE aware application in the form of either CFBundles (or for legacy puposes, code fragments) on the Macintosh, or DLLs (Dynamically Linked Libraries) on the Windows Platform. A basic understanding of shared libraries and Platform specifics will help you in plug-in development, but is not required.

For information on platform specific architecture consult the following URLs.

Windows:
http://msdn.microsoft.com/library/default.htm (search for “DLL”)

Macintosh:
http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFBundles/CFBundles.html
http://developer.apple.com/techpubs/macos8/mac8.html (search for “Code Fragment Manager”)

Digidesign Plug-In Architectures

AudioSuite

AudioSuite plug-ins are non-realtime/semi-realtime, or destructive processes (ignoring Pro Tools undo function.) These plug-ins allow you to perform file-based processing. In other words, the processing is done only on prerecorded blocks of audio. They generally work in two ways. The first way is to selectively apply the processing algorithm to a particular audio track(s). This is known as “destructive” processing, because the original audio track is replaced by the new processed audio track. There are no limitations governing the amount of time required to process a track in this manner. For this reason, Audio Suite Plug-ins are considered to be non-realtime. Audio Suite plug-ins also have a second optional mode in which they can run. This is referred to as Preview mode. The Preview feature allows you to monitor the audio processing applied to a track in real-time, or semi-real-time. Because this is a realtime process, it is not applicable to all types of file based processing. You may elect not to support this mode if your algorithm does not lend itself to realtime processing. Preview mode is implemented in a non-destructive manner. When running in preview mode the processing done is for auditioning purposes only, and no replacement of audio tracks occurs. Audio Suite Plug-ins are coded to execute entirely on the host processor, and require no Motorola 56K DSP programming.

Real-Time AudioSuite (RTAS)

An RTAS plug-in is basically an extension of an existing AudioSuite plug-in. RTAS plug-ins behave similarly to traditional TDM plug-ins, with the exception that they are implemented entirely on the host processor. You get the benefits of non-destructive realtime processing without the hassle of DSP code development. However, the system CPU can quickly become taxed, since all signal processing is done on the host. These plug-ins are also automatable, which means that control movements can be dynamically recorded and played back with the audio track. Automation also allows you to control your plug-in from external control surfaces.

TDM

TDM plug-ins are realtime non-destructive processes which can be used in live recording situations, as well as on pre-recorded tracks. One of the biggest benefits of TDM plug-ins is that the processing is done entirely on DSP chips. This prevents the host CPU from being tied down with expensive signal processing tasks. The reason TDM plug-ins are termed non-destructive is that audio data can only be effected while there is an instance of the plug-in open on the track. There is no replacing of audio, as in the case with Audio Suite. You can however, achieve a permanent result by bouncing the audio track to disk while the plug-in is inserted on the track. Like RTAS, TDM plug-ins are automatable. Developing a TDM plug-in requires extensive knowledge of the Motorola 56K DSP architecture.

Getting Started With the Source Code

The SDK Source Tree

The source tree heirarchy of the Plug-In SDK might seem a bit strange at first contact. It mimics the strange hierarchy of our internal source control system’s directory structure. Despite being slightly confusing at first, it greatly expedites the creation of a new SDK. This is especially critical to us and to you during new development cycles -- when both are pressing forward to release new products. Here’s a simplified heirarchy with the most important directories.

- **`AlturaPorts`** (Altura is the vendor of the Mac2Win cross-platform porting library.)
    - **`DigiPublic`** Misc. files. Most important is DSPIncludes, required for building DSP code

- **Fic** Header and interface files for DAE.
- **Libs** Mac library files needed for plug-in development.
- **TDMPlugIns** A misnomer. This folder contains all the sample plug-ins, RTAS, AS, or TDM, and the Plug-In Library.
    - **common** Some DSP code build scripts, precompiled header files, & misc.
    - **DSPManager** COM-like interfaces required by MultiShell plug-ins.
    - **PACEProtection** supporting files for plug-ins that use PACE copy protection.
    - **PluginLibrary** The core classes required for building a plug-in, typically compiled into a .lib file for linking into the sample plug-ins or yours.
    - **SamplePlugIns** contains all example plug-ins that Developer Services provides.
        - **_AllSamplePlugIns** Resources that are common to all and/or multiple sample plug-ins.
        - **56kTdm2DmaExamplePlugIn** Sample plug-in. Demonstrates an advanced DMA technique for TDM plug-ins. See the "Surround Downmixer" chapter of this document.
        - **Microbe_SamplePlugIns** A suite of sample plug-ins that use MIDI.
            - **Microbe** Implements a volume control controlled via MIDI in RTAS and Non-MultiShell TDM. See the "Microbe" chapter of this document.
            - **MicrobeSampler** Implements a very basic sampler plug-in in RTAS. Demonstrates how to write an efficient instrument plug-in (MIDI in, audio out) on the Pro Tools platform. Also demonstrates using MIDI outputs (available in PT 7.2 and later)
        - **ReverseDoubleHalf** AudioSuite sample plug-in. Exercises non-realtime, non-linear functionality of AudioSuite.
        - **SampleClick** Another DirectMidi sample plug-in. Implements a click track and demonstrates using Midi Beat Clock.
        - **SimplePlugin** A very simple plug-n that demonstrates the very minimum required implementation for an RTAS plug-in.
        - **Template_SamplePlugIns** A suite of non-MIDI sample plug-ins that implement volume control.
            - **Template** Demonstrates RTAS, AS, MultiShell TDM and Non-MultiShell TDM, all in the same binary.
            - **Template_DMA** Non-MultiShell TDM-only, HD|Accel-only. Uses an advanced technique for polling the host port using the DMA controller for fast host-to-DSP I/O.
            - **Template_NoUI** Demonstrates how a plug-in can achieve independence from the Digi UI framework. Useful for developers who want to use their own UI classes. Currently available on Windows only.
- **DSPTools** 56k DSP assembler and scripts.

Sample Plug-Ins To Start With

The Plug-In SDK contains several sample plug-ins that should make good starting points for your development.

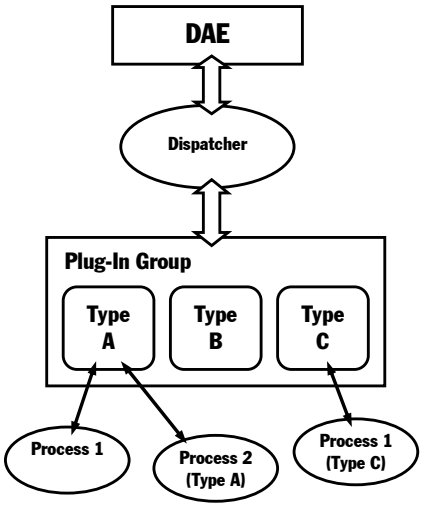
If youre developing for...	Then, we recommend you start with...
AudioSuite	ReverseDoubleHalf
RTAS non-MIDI	Template or SimplePlugIn
RTAS w/MIDI	Microbe
MultiShell TDM	Template
Non-MultiShell TDM	Template
Non-MultiShell TDM w/ MIDI	Microbe
RTAS Instrument (MIDI in, audio out)	MicrobeSampler
MIDI Outputs	MicrobeSampler
Plug-ins that use non-Digi UI framework	Template_NoUI

How the Plug-In Environment Works

This section conceptually explains how the Digidesign Plug-In Environment operates. It generally applies to all three of the plug-in architectures, but the specific differences between them are explained in each corresponding chapter. Issues such as the software structure and dispatching calls to objects within this structure are discussed. An understanding of the internal workings of the plug-in environment is needed in order to create your own custom plug-in. Detailed descriptions of classes and methods that makeup the environment can be found in the Function Reference.

Overall Structure: Interaction Between DAE and Plug-Ins

A DAE plug-in is implemented as a Shared Library on MacOS, or a DLL on Windows. When DAE launches, it will search the "`Plug-Ins`" folder, then using the Code Fragment Manager, will load, initialize and catalog any plug-in modules that it happens to find there. Once initialization is complete, DAE can then communicate with a plug-in module by passing it information that the plug-in must react upon either by performing an event or passing back needed information. As a result, developers that produce applications that are DAE aware (i.e., in our case Pro Tools) now have the capability of adding considerable functionality to their application via third party plug-in effects.



The DAE Plug-In Structure

There are three basic C++ objects used in the plug-in environment. As a developer, you will inherit from these two or three base classes, then extend their functionality to complete your plug-in. Conceptually, these three classes can be grouped into three "levels." These three classes or levels are the Group Level, Type Level and Process Level.

Note: In previous SDK's the Group and Type levels were referred to as the ProcessGroup and ProcessType. We found this excessive use of the word "Process" to be confusing and offensive. Thus, we have tightened up the nomenclature by removing the word "Process" where it is extraneous.

Let's examine these three classes in a bottom-up approach.

Process Level

An object created on the Process Level is called a Process object. A Process is probably most closely related to what a Pro Tools user's conception of a plug-in is. A Process is instantiated for every plug-in insert made within a session, and manages all the information related to that insert.

Typically, Processes also have a graphical user interface through which the user can manipulate the audio effect; thus, the Process is responsible for relaying these parameter changes to the algorithm. In the RTAS system, this algorithm is also encapsulated within the Process. For a TDM plug-in, the Process must communicate algorithm parameters to a DSP Process that is created on the TDM hardware, simulateously with the host Process. The number of Process objects that can be instantiated is limited by the available hardware in the TDM system.

Type Level

An object created on the Type Level is called a Type object. The Type Level serves two functions. It provides a description of the Process, and manages all of its Processes as they are instantiated. The description of the Process includes such things as the plug-in's name, its I/O configuration, hardware platform (host or TDM), and other varied properties. The management functionality of the Type Level is pre-built into the class.

Group Level

An object created on the Group Level is called a Group object. There is only a single instantiaion of the Group for a given plug-in. This object is responsible for returning general information about the plug-in, such as manufacturer identification and how many different effects the plug-in can do. A Group object is also responsible for creating the Type objects and managing a list of all Type objects that a plug-in owns.

A single group level object gets created when the DAE aware application calls your plug-in's initial entry point `NewPlugIn`. This, in-turn, results in a call to the static method `CProcessGroup::CreateProcessGroup()`. The group object remains in scope until all instances of your plug-in have been removed from the parent application, and `ClosePlugIn` has been called to destroy it. The main purpose of the Group object is to create and maintain a list of Type objects.

How Plug-In Objects are Identified

■ Since there is only one instantiation of a Group object in a plug-in, a pointer to this object is stored in a global variable. After DAE has interfaced with the Code Fragment Manager, the Group object can be accessed by this global variable.

■ Every Type object is identified with a unique `OSType` that is defined by the plug-in. When a plug-in is first instantiated, DAE will request the `OSType` of all the Type objects. Then, in subsequent communications with the plug-in, it will pass in this identifier to indicate the destination Type for the given request.

■ Every Process, created by and managed by a unique Type, is identified with a unique index. The plug-in is responsible for generating this unique index and returning it to DAE when a new Process instantiaton is requested. DAE then uses this index to reference the appopriate Process in subsequent communications.

How Methods are Dispatched

The above sections discussed some of the different types of objects that are instantiated in the plug-in Environment. DAE, at any time, can call any one of these objects to request information or tell it to perform an action. The plug-in environment has a unique way of dispatching these calls ensuring that they end up at the desired object. Understanding this dispatching mechanism is necessary in order to understand the overall interaction between all objects instantiated in the plug-in Environment.

All DAE requests that are passed through the Dispatcher are intended to end up at one of the three object types: the Group object, a Type object or a Process object. To identify the Dispatcher method to be called, DAE uses a selector. After determining the method, `Dispatcher.cpp` passes the call along with parameters to the intended method in the Group object.

If the receiving method in the Group object is designated to dispatch the call to a Type object, it will use the `OSType` information found in the parameter list to identify the intended Type object and dispatch the call.

If the receiving method in the Type object is designated to dispatch a call to a Process object (i.e. a running Process), it will use the index information found in the parameter list passed to identify the intended Process object and dispatch the call.

All that has to be remembered here is that an `OSType` represents a unique Type and an `OSType`/index pair represents a unique Process.

Reiterating, as this SDK has loved to do, DAE interacts with your plug-in by dispatching function calls to the entry points defined in the dispatcher. When a call is dispatched in this manner, it has one of three possible destinations, Group, Type or Process. The group object is global to the plug-in, and all dispatched calls are filtered through it. If a call is intended to trickle down to the type or process level, it will contain identifying information in the function arguments. When a call is targeted at a particular Type, a Type ID (`OSType`) argument will be present. If the Target is a particular Process object, additionally there will be a Process Index (`long`) argument. The plug-in library takes advantage of this identification information, to insure that all calls eventually arrive at the appropriate level.

Interfacing to the Outside World

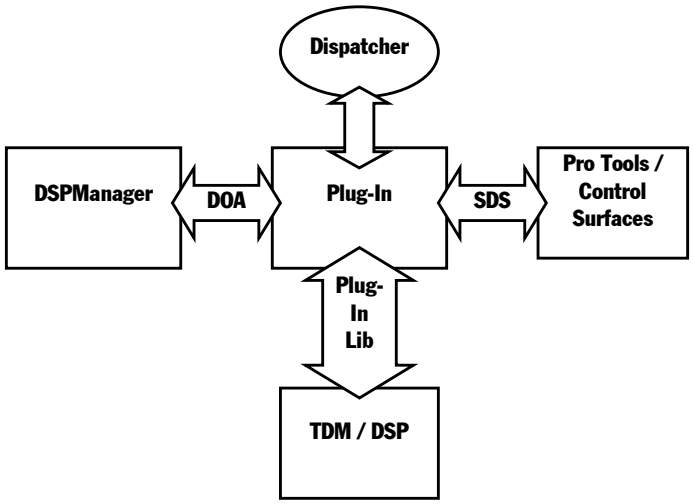
Since a plug-in is a shared library, and not a stand-alone application, it must communicate with other executing code in order to be useful. There are several ways in which this occurs.

■ You will find that the majority of the interaction your plug-in has with the external world occurs through the entry points defined in `Dispatcher.cpp`. Communication in this manner tends to be one-way, where DAE calls one of your library routines, and your plug-in responds to its request. Typically responding to a request requires your plug-in to perform some specialized task and/or return information.

■ A second way in which TDM plug-ins communicate externally is through the DOA (Digidesign Object Architecture) interface. This is a “Microsoft COM”-like interface used for communication between the plug-in and the DSP Manager. Currently this interface is not used for other purposes, however that may indeed change over time. This COM interface is mostly transparent to the developer.

■ A third way in which a plug-in can communicate with the outside world is through Shared Data Services (SDS or Token System). This is a mechanism that allows Pro Tools to share parameter information with external HW/software modules. Plug-ins use the token system to automate their controls. Using the SDS mechanism you will be able to have your plug-in send out and listen for specific tokens. The SDS system is utilized inside the Plug-In Library and is transparent to the developer.

■ Lastly, through API calls of the Plug-In Library, a plug-in is able to communicate with the DSP in the TDM system.



The Life of a Plug-In

When you launch Pro Tools, it immediately begins loading DAE. DAE is pretty smart, it knows if anything has changed within the Plug-ins folder. If nothing has changed since the last time it was launched, DAE relies on information cached from the previous launch to speed through the plug-in loading process. If, however, something has been added, removed, or updated since the last launch, DAE must check each item in the entire Plug-in folder to determine what different Types are available. DAE then hands this information back to Pro Tools so that Pro Tools knows how to set up its AudioSuite and plug-in insert menus.

Let's assume that you have just built a new plug-in for the first time and moved it into the Plug-ins folder. What happens when you launch Pro Tools? Pro Tools launches DAE. DAE recognizes that the state of the Plug-ins folder has changed. It goes through each item in the folder. When it reaches your new plug-in, it must determine what process Types are available. To determine this, DAE must create each of these Type objects.

DAE creates a new plug-in by calling the plug-in's exported `PlugInInitialize.cpp` routine `NewPlugIn()`. `NewPlugIn()` calls `CreateProcessGroup()` which makes the call to construct the group object. This method might look something like:

```
CProcessGroupInterface*  CProcessGroup::CreateProcessGroup(void)
{
     return new CYourPlugInGroup;
}
```

The Group constructor creates the object and defines the manufacturer name and ID, plug-in name and version, and any supported gestalt properties. A typical Group constructor might look like:

```
CMyGroup::CMyGroup(void)
{
     DefinePlugInNamesAndVersion("My Plug-In's Name\nMy Plug-In\nMine", 1);
     DefineManufacturerNamesAndID("Digidesign\nDigi", 'Digi');

     AddGestalt(pluginGestalt_SupportsDeckChange);     // TDM & RTAS pi types are interchangeable
     AddGestalt(pluginGestalt_IsCacheable);            // Plug-in will be cached for faster DAE loading
}
```

The Group object is now created and both calls return. Next, `NewPlugIn()` attempts to initialize the object by calling the Group object's `Initialize()` method.

```
void CMyGroup::Initialize (void)
{
     CEffectGroup::Initialize ();  // Always call inherited first

     // 1. Register View classes.
     // 2. Perform any other miscellaneous global initializations
}
```

Calling the inherited `CEffectGroup::Initialize()` does a some important internal setup. Most importantly, it triggers a call into the Effect Layer's `CreateEffectTypes()`.

```
void CMyGroup::CreateEffectTypes(void)
{
     // 1. Generate all Type objects
     // 2. Attach Process classes to each
     // 3. Register each Type with the Group via AddEffectType()
}
```

After this, the `NewPlugIn()` call returns. The plug-in is now in a complete, but inactive state. DAE can now, via the dispatcher, request any information from the plug-in about its configuration. Such request include the plug-in's name, manufacturer, the number of Types it supports, the name of each Type, etc.

After it has gathered all the information it needs, DAE calls the `PlugInInitialize.cpp` method `ClosePlugIn()`. This subsequently calls the destructors for all the Type objects, then the Group. DAE now hands of all the necessary information to Pro Tools. Pro Tools can build its AudioSuite and plug-in insert menus and finish loading. After this process has completed for all plug-ins, the user can start working with the Pro Tools session.

Summarizing, here is the order of setup for a new plug-in:

1 The exported function `NewPlugIn()` is called in the plug-in shared library or DLL.

2 `NewPlugIn()` creates a new Group object, which subsequently invokes the Group's constructor.

3 The Group method `CreateEffectTypes()` is callled.

4 The body of the Group's `Initialize()` method is executed.

5 `NewPlugIn()` returns, and DAE queries about the plug-in's configuration.

6 `ClosePlugIn()` is called, and Type objects and Group object are deleted.

Chapter 3: Using the Effect Layer

Historically, developers have found the Plug-In SDK to be overwhelming at first. There is a large learning curve. Over the years, the plug-in Library has grown to be somewhat bulky, confusing, and tough to navigate. In all its complexity, it is really quite powerful. However, in response to Plug-In Library's drawbacks, the Effect Layer was created. All the sample plug-ins utilize the Effect Layer.

For those familiar with traditional DAE plug-in development, let's quickly examine how using the Effect Layer differs and what advantages it has. First off, the Effect Layer implements all generic functionality that plug-ins should require and reduces as much as possible the workload of the developer. For example, the Effect Layer handles the "Chunk" methods for you -- saving and restoring all the controls of a plug-in automatically. Controls must be implemented using the Control Manager for this functionality. The Effect Layer also handles deck swapping plug-ins (converting RTAS to TDM, and vice versa) in a generic manner. In general, the Effect Layer provides a more streamlined and coherent API over prior SDK releases. Another benefit of the Effect Layer is that the generalization and centralization of plug-in code should help plug-in developers reduce bug counts and maintenance time.

Note that plug-ins may still be developed without using the Effect Layer (as many of our longtime developers will attest); however, starting with the Pro Tools 6.4 release, Developer Services is no longer providing direct support in the form of documentation and sample code for non-Effect Layer plug-ins. Supporting two separate plug-in APIs was proving to be too distracting from our larger goal of providing the best possible tools, SDKs, and support materials to our developer community. Note also that Digidesign internally has been using the Effect Layer for all new plug-in development beginning with the 6.4 release.

This chapter is only intended to be a general and brief overview of the Effect Layer's class structure. The details involved in utilizing the Effect Layer will be left to the more demonstrative sample plug-in descriptions presented later in the SDK.

The Effect Layer Classes

The Effect Layer introduces a slightly different development paradigm than previous plug-in SDKs. The same Group-Type-Process structure still exists. However, the conceptionalization of the Type level has changed. The usual inherting and overriding of this class is usually not necessary. Instead, the Type object can be thought of as simply a descriptor class which provides the layout and configuration of the Processes it is capable of instantiating. Instead of inheriting from this class, you instantiate a generic object of the approriate Type class, then manipulate its configuration via a set of methods. After this newly formed description is finished, it is passed to the Group object.

In addition, at the Group and Process levels, the Effect Layer makes considerable use of C++'s capacity for multiple inheritance. By mixing core functionality classes in with support classes, a complete class is created from which the plug-in can be built.

So, the first step in building a new plug-in with the Effect Layer is to decide on the classes from which you should inherit.

Group Level

The following table outlines which base class your Group level should inherit from, dependent on the platforms your entire plug-in will support. For MIDI support, use multiple inheritance to mix-in the additional `CEffectGroupMIDI` class.

Plug-In Platforms To Be Supported	Without MIDI	With MIDI
AS/RTAS only	CEffectGroup	virtual CEffectGroup, CEffectGroupMIDI
MultiShell (MuSh)	CEffectGroupMuSh	CEffectGroupMuSh, CEffectGroupMIDI
MuSh with RTAS/AS	CEffectGroupMuSh	CEffectGroupMuSh, CEffectGroupMIDI
Non-MuSh	CEffectGroupTDM	CEffectGroupTDM, CEffectGroupMIDI
Non-MuSh with RTAS/AS	CEffectGroupTDM	CEffectGroupTDM, CEffectGroupMIDI

Process Level

The following table outlines the classes from which to inherit when forming your Process level classes. The Effect Layer imposes a specific inheritance tree structure that should be used:

First, inherit from the core functionality class `CEffectProcess`, or from `CEffectProcessMIDI` for MIDI support. Implement all generic plug-in code that would be universal and independent of the intended platform, e.g. controls, GUI code, user input, etc. This results in the class that we will call, for the sake of discussion, `CYourGenericProcess`.

Next, for each platform this Process is intended to run on, multiply inherit from `CYourGenericProcess` and the `CEffectProcess`*`Platform`* support class. This mixes-in the specific AS, RTAS, MuSh, or Non-MuSh TDM support to your central plug-in code. Within this new class, you can finish creating the Process, implementing anything specific to platform, namely the algorithm, or in the case of TDM, the communication protocol with the DSP.

Process Platform	Without MIDI	With MIDI
Typeless/Generic Process	CEffectProcess	CEffectProcessMIDI
AudioSuite (or AudioSuite-adaptor style RTAS*)	CYourGenericProcess, CEffectProcessAS	CYourGenericProcessWithMIDI, CEffectProcessAS
RTAS	CYourGenericProcess, CEffectProcessRTAS	CYourGenericProcessWithMIDI, CEffectProcessRTAS
MuSh TDM	CYourGenericProcess, CEffectProcessMuSh	CYourGenericProcessWithMIDI, CEffectProcessMuSh
Non-MuSh TDM	CYourGenericProcess, CEffectProcessTDM	CYourGenericProcessWithMIDI, CEffectProcessTDM

* "AudioSuite-adaptor style RTAS" denotes an RTAS plug-in that uses the AS classes, but that runs in real-time. It is useful for developers who start with an AudioSuite plug-in and want to quickly port to RTAS, but don't need the advanced features offered in the "true" RTAS. These features include access to the HW Buffer size and "callback count", among other RTAS-engine specific parameters. Currently the Microbe and SampleClick sample plug-ins in the SDK use the Adaptor-style RTAS, whereas the Template and MicrobeSampler plug-ins use "true" RTAS.

Type Level

Again, the Type level is not intended to be inherited from, except in special cases. Instead, the Type classes should only be instantiated, typically within the Group level's `CreateEffectTypes()` method. Using the interface methods of the Type, a description of the Process that it represents is generated and

passed to the Group via the `AddEffectType()` call. Both the **Sample Plug-In Code** and the **Plug-In Library/Effect Layer Reference Guide** provide detailed information on the Type object's interface methods.

The following table outlines the four Type level classes that can be used, dependent on the platform the Process will run on.

Type Platform	Effect Layer Class
AudioSuite	CEffectTypeAS
RTAS	CEffectTypeRTAS
MuSh TDM	CEffectTypeMuSh
Non-MuSh TDM	CEffectTypeTDM

Building a Non-MultiShell or MultiShell Effect Layer

The Effect Layer supports both MultiShell and regular TDM plug-ins. To select between these two modes, in addition to selecting the appropriate classes, a `#define` switch must be used. `#define EFFECT_LAYER_IS_MUSH` should be placed in your precompiled headers if a MultiShell-enabled plug-in is desired.

Saving & Restoring Custom Data

There are many cases when the state of a plug-in must be captured or restored by DAE -- e.g., when the user saves settings, when a session is saved, or when a plug-in is converted from TDM to RTAS. Plug-in state information is stored in *chunks*. Chunks are arbitrary sized data structures with a standard, fixed-size header. The Effect Layer automatically handles the chunk to save and restore the state of the controls for a plug-in. In addition, the developer may add custom chunks to a plug-in. To specify a custom chunk, in the Process's constructor, use the call `AddChunk()`. This call can take three parameters: an `OSType` ID of the chunk, a string description of the chunk's contents, and a byte count for the size of the data. For example,

```
AddChunk('anID', "Reverb Coefficients", 16);
```

Currently, Pro Tools does not make use of the description string, but it doesn't hurt to add it. Alternatively, the chunk size may be left undefined, implying that the chunk size is dynamic and/or will be determined at run-time.

```
AddChunk('anID', "Reverb Coefficients");
```

In this case, it will be necessary to override the method `GetChunkSize()` to specify the chunk size during runtime.

The implementation of two more calls are required to finish the chunk system: one to restore the state of the plug-in, the other to capture the state of the plug-in. `SetChunk()` is invoked when DAE is passing in a chunk that the plug-in should restore. `GetChunk()` is called when DAE is requesting a chunk from the plug-in. DAE will always call `GetChunkSize()` prior to `GetChunk()` so that it can preallocate a chunk buffer of the required size.

Note: As described in the next section, a chunk with the ID 'midi' is only saved and restored with the session, and is not included when the user saves settings. If there is session specific information that should not be included in a preset, it is possible to include this information in the 'midi' chunk.

The **Plug-In Library/Effect Layer Reference Guide** illustrates how to implement these methods, in addition to the next section.

ChunkDataParser

The `ChunkDataParser` is a helper class that provides a generic, abstracted, and endian-independent data repository that can be used for manipulating chunk data.

Think of the `ChunkDataParser` object as a container for chunk data. By utilizing the parser's interface methods, a chunk can be loaded into or retrieved from the container. Data values can then be added to or retrieved from the chunk using the parser's set of "Add" and "Find" methods. Every data item that is stored in the chunk has an associated unique string name that is used to reference the value. Currently, the `ChunkDataParser` handles four data types: `double`, `float`, `SInt32`, and `SInt16`. Let's now look at the parser's interface methods.

■ `LoadChunk(SFicPlugInChunk *chunk)` is obviously used to load an initialized chunk into the repository. An important thing to note is that the parser does not maintain the underlying header information of the chunk. However, in typical usage this isn't critical since the header is already preset at the begining of a `GetChunk()` or `SetChunk()` Process call.

■ `GetChunkData(SFicPlugInChunk *chunk)` copies the chunk data into a preallocated chunk. The header information of the chunk, except for the chunk size, is left unaffected.

■ `SInt32 GetChunkSize()` calculates and returns the required buffer size in bytes to hold the chunk currently represented within the container.

■ The following methods are all used to add or retrieve data values from the container. Data added to the container is permanent and is not removed using a "Find" method. If the value named with the string `char *name` is contained in the chunk data, the "Find" methods return true and it copies the value into the variable pointed to by `*value`.

```
void AddFloat(const char *name, float value)
bool FindFloat (const char *name, float *value)

void AddDouble(const char *name, double value)
bool FindDouble(const char *name, double *value)

void AddInt32(const char *name, SInt32 value)
bool FindInt32(const char *name, SInt32 *value)

void AddInt16(const char *name, SInt16 value)
bool FindInt16(const char *name, SInt16 *value)
```

■ `void Clear()` removes all data values from the container.
■ `bool IsEmpty()` returns true if the container currently holds no data values.

Using the ChunkDataParser

The Effect Layer class `CEffectProcessMIDI` used to utilize the `ChunkDataParser` to store and retrieve the state of the antiquated OMS MIDI system. Although the OMS system is no longer in place, its implemention is an excellent example of how to deal with custom chunks.

First off, the chunk must be declared in the constructor using the `AddChunk()` call:

```
CEffectProcessMIDI::CEffectProcessMIDI(void)
     : mMidiChunkSize(0)
{
     AddChunk(EffectLayerDef::MIDI_CHUNK_ID, "MIDI Connection Data");
}
```

Since a fixed chunk data size was not defined, the method `GetChunkSize()` must be overridden and implemented.

```
ComponentResult CEffectProcessMIDI::GetChunkSize(OSType chunkID, long *size)
{
     if (chunkID == EffectLayerDef::MIDI_CHUNK_ID) {
          if (mMidiChunkSize == 0)
               mMidiChunkSize = BuildMidiChunkData();
          *size = mMidiChunkSize;
          return noErr;
     } else
          return CEffectProcess::GetChunkSize(chunkID, size);
}
```

Here, we catch our custom chunk ID, calculate its size, and cache this value in `mMidiChunkSize`. If the `chunkID` is not one of our custom chunks, we *must* pass the chunk onto the inherited class.

Since this chunk size is constant, we don't want to rebuild the chunk every time `GetChunkSize()` is invoked. This is why we store the value for subsequent calls. Now, the helper method `BuildMidiChunkData()` looks like:

```
SInt32 CEffectProcessMIDI::BuildMidiChunkData()
{
     mMidiChunkParser.Clear();  // Reset the container

     mMidiChunkParser.AddInt16("Unique OMS ID", fNodeUniqueID);
     mMidiChunkParser.AddInt32("OMS Node ID", fNodeID);
     return mMidiChunkParser.GetChunkSize();
}
```

This method simply stashes new chunk data within the `ChunkDataParser` member of the Process class, `mMidiChunkParser`. As you can see, two data values that deal with the MIDI system, `fNodeUniqueID` and `fNode`, are stored in the chunk each with unique names. The return value of this method is the total size of the chunk.

`GetChunk()` also relies on the `BuildMidiChunkData()` method:

```
ComponentResult CEffectProcessMIDI::GetChunk(OSType chunkID, SFicPlugInChunk *chunk)
{
     if (chunkID == EffectLayerDef::MIDI_CHUNK_ID) {
          BuildMidiChunkData();
          return mMidiChunkParser.GetChunkData(chunk);
     } else
          return CEffectProcess::GetChunk(chunkID, chunk);
}
```

Again, we catch only our custom chunk ID(s) and pass all others onto the inherited class. The parser's `GetChunkData()` method is used to move the chunk data into the empty incoming chunk passed in by DAE.

Note: Take care not to perform time intensive tasks, such as disk I/O, within the GetChunk() method. This method is intended for saving plug-in settings into a chunk. **Any extra work performed here could hold off the Pro Tools GUI without notifying the user until the work is completed.** Such a scenario would result in the user seeing the “busy” mouse cursor (hourglass/spinning wheel) every time GetChunk() is invoked; GetChunk() is called on auto save, when the user saves plug-in preset settings, or on session close.

Lastly, we need to implement the `SetChunk()` method:

```
ComponentResult CEffectProcessMIDI::SetChunk(OSType chunkID, SFicPlugInChunk *chunk)
{
     if (chunkID == EffectLayerDef::MIDI_CHUNK_ID)
     {
          OMSSignature        nodeID;                              // Used to sign into OMS
          OMSUniqueID         uniqueID;                            // Used to sign into OMS

          mMidiChunkParser.LoadChunk(chunk);

          if (mMidiChunkParser.FindInt16("Unique OMS ID", reinterpret_cast<SInt16 *>(&uniqueID)) == false)
               return EffectLayerDef::WEAK_ERROR_CODE;
          if (mMidiChunkParser.FindInt32("OMS Node ID", reinterpret_cast<SInt32 *>(&nodeID)) == false)
               return EffectLayerDef::WEAK_ERROR_CODE;

          if (uniqueID != fNodeUniqueID)
```

```
        {
            // [Stuff deleted] Process chunk values here...
        }

        return noErr;
    } else return CEffectProcess::SetChunk(chunkID, chunk);
}
```

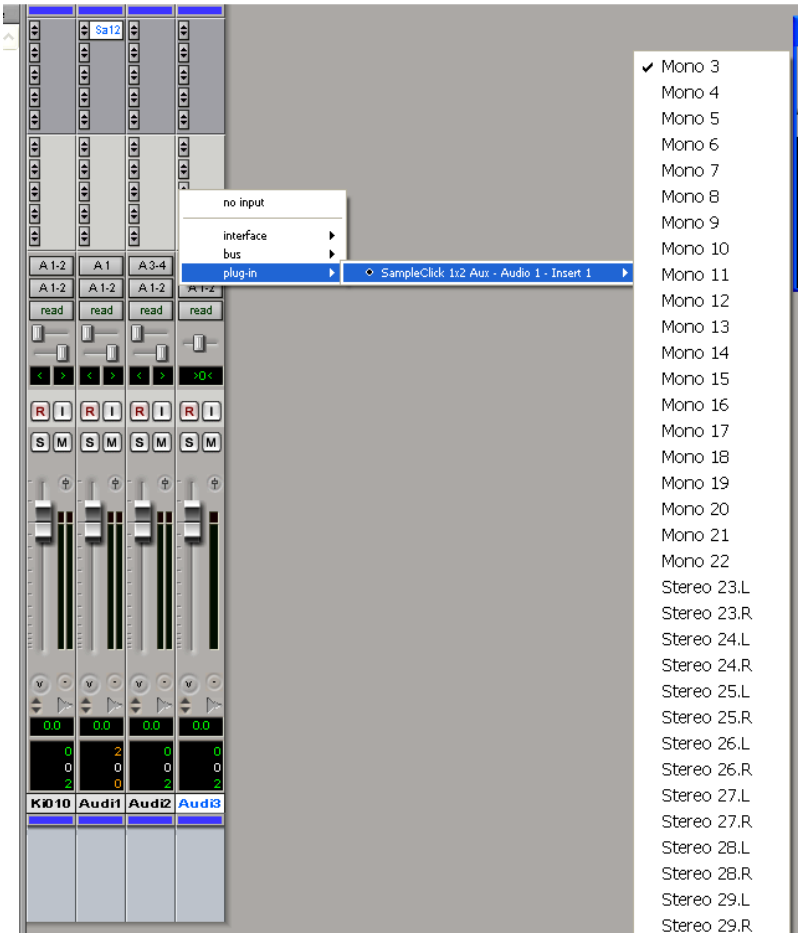
Here, we load the valid incoming chunk into the `ChunkDataParser` container using the `LoadChunk()` method. Then, the two expected values are extracted using their string names and the appropriate *`Find`* method. If either of the values is missing from the chunk, an error is returned. Lastly, the necessary processing of the chunk values is done.

Chapter 4: Plug-In Features

Digidesign plug-ins offer users a rich set of features. To make sure your plug-ins are as feature-complete as possible, where applicable you should implement all of the features presented in this chapter.

Feature: Auxiliary Output Stems

Pro Tools has the capability to show and route multiple "auxiliary" outputs from a plug-in to other tracks. These are known as Auxiliary Output Stems (AOS), a stem referring to one set of outputs. A stereo stem contains two outputs, left and right, and a mono stem contains one output. The outputs will appear in the input assignment pop-up menu of each track under the category "plug-in" (as shown below). A plug-in has the option to take advantage of this feature.



Feature Subtleties

Auxiliary Output Stems are for both TDM and RTAS plug-ins, and work on both TDM and LE systems. Here are some more important subtleties about this feature you will want to know:

Mono and Stereo Stems Only Only mono and stereo stems are available as auxiliary outputs.

Displayed Text Strings for Aux Outputs Since it is possible to use multiple copies of the same multi-output plug-in, a naming scheme has been established to allow for easy identification of an aux output's source plug-in. The plug-in submenu for a track's input selector will show the following information:

- Plug-in name
- Track name the plug-in is instantiated on
- Insert letter the plug-in is instantiated on (the word "Insert" followed by a letter)

Names will be separated by a hyphen (-). For example, if you had instantiated a multi-output plug-in called "Sampler" on the first insert of a track called "Music", the plug-in sub menu will show the following string: "Sampler – Music – Insert A."

Note that the screenshot shows insert numbers instead of letters. "Insert 1" is shown instead of "Insert A". This is because the screenshot was taken from a beta version that does not yet reflect this. The finalized version will have letters, not numbers.

Non-Dynamic Allocation of Aux Outputs The aux outputs cannot be added and removed from the system dynamically though they can be made inactive by the user. The total number of aux outputs, stem types, names, paths, and ordering are defined only once by the plug-in.

No Support for Sidechain Inputs Plug-in aux outputs are not available from the sidechain input popup menu in other plug-ins. Users will not see the "plug-in" submenu when clicking on a plug-in sidechain popup.

Moving a Plug-In to a Track Assigned with its Aux Output If a multi-output plug-in is dragged to a track that has one of the plug-in's aux outputs assigned as track input, the input will be made inactive.

Multiple Aux Output Assignments You may route any plug-in aux output to any disk or aux track input if it matches the track's input width. Mono tracks can only take mono aux outputs for its input and stereo tracks can only take stereo aux outputs for its input. This includes routing the same plug-in aux output to multiple track inputs. Pro Tools automatically creates mono sub paths from stereo plug-in aux outputs so that the individual channels of a stereo aux output stem, left and right, can be assigned to a mono track input.

Multi-Mono Not Supported There cannot be any multi-mono multi-output plug-ins. If a mono plug-in instance offers multiple outputs it cannot support multi-mono.

TDM Time Slots Usage Multi-output TDM plug-ins will consume a TDM time slot for each aux output channel, when that connection is made. For example, instantiating a stereo TDM plug-in with one stereo main and six mono aux output stems will initially consume two TDM time slots for its main outputs. Each aux output connection made will require an additional TDM time slot. Multi-output RTAS plug-ins also consume TDM time slots, but do not require a set amount of slots.

TDM/LE Disk Voices Usage On TDM, multi-output TDM plug-ins have no need for additional TDM disk voices. Multi-output RTAS plug-ins need a TDM disk voice for each aux output assigned to a TDM track input. The exception is that assigning the same plug-in aux output to additional track inputs does not consume more voices. On LE, plug-in aux outputs consume a certain amount of CPU power, but don't impact the number of available LE disk voices for playback and recording.

Low Latency Monitoring If Low Latency Monitoring is activated, all the aux outputs of bypassed plug-ins are bypassed. The track input display of assigned aux outputs will not change.

Record-Enabling TDM Audio Tracks When using RTAS plug-ins on a TDM disk track, record-enabling that same track also bypasses the RTAS plug-in. Consequently, all aux outputs are bypassed. Track input display of assigned aux outputs will not change.

Plug-In Implementation Responsibilities for Auxiliary Output Stems

If a plug-in is going to utilize the AOS feature, it will be responsible for a few details that are summarized below:

Aux Output Paths The plug-in is responsible for the definition of valid aux output paths. This definition includes the total number of outputs, the desired order of stereo and mono paths. Pro Tools will query each plug-in for available valid paths and populate its track input selector popup menus accordingly.

Aux Output Path Order The plug-in is responsible for specifying the type and name of each of its aux output paths . A plug-in decides whether the aux outputs are all stereo, all mono, "X" stereo outputs followed by "Y" mono outputs, or some other combination. Pro Tools lists each output in the order given by the plug-in. If mono and stereo paths are interleaved the input popup menu of the mono tracks keeps that order and breaks the stereo paths into their respective left and right sides using ".L" and ".R" suffixes.

Aux Output Names A plug-in is responsible for giving meaningful names to aux outputs. Names are only defined once, so they will stick. At the very least, individual outputs should be labeled "Output xx", where "xx" is the aux output number as it is defined in the plug-in. The output name should also include the words "mono" and "stereo" to support when users are looking for an output with a specific stem format.

Aux Output Numbering The plug-in is responsible for defining the lowest available aux output number. Plug-ins should base this number on the width of the plug-in's main outputs. For example, when using a stereo instance of a sampler the first aux output should be #3, when using a 5.1 instance of the sampler the first aux output should be #7, etc. This is to keep the numbering scheme inside of the plug-in and in Pro Tools consistent. From Pro Tool's perspective, plug-ins typically enumerate all available outputs and do not differentiate between main and aux outputs. The first "N" outputs are used for the main outputs, and all the remaining outputs are available for aux output paths.

Separate Multi-Output Plug-In Process Type Plug-in developers are encouraged to offer both "regular" and "multi-output" versions/types of any multi-output capable plug-in. We strongly suggest this to conserve resources and to keep the user's workspace as uncluttered as possible. Users can choose to use the regular version/type for plug-ins they don't need aux outputs for. Multi-output versions can be created as separate process types so that there need not be separate binaries. Such additional process types will be listed in the plug-in menu next to their regular version siblings. They should be nominally distinguished by appending phrases like "multi-output" to the plug-in name, for example.

Note that when moving sessions between different PT systems, multi-output process types will NOT be automatically converted to regular process types if multi-output types are not available. This relation must be defined manually by the plug-in developer via RelationID's.

No Multi-Mono Implementations A plug-in is responsible for not having multi-mono enabled if it utilizes auxiliary outputs stems. Auxiliary output stems will not work in multi-mono enabled plug-ins. Multi-mono is automatically disabled for AOS in the Effect Layer.

Implementing Auxiliary Output Stems

Here are the first steps for implementing AOS in a plug-in; these steps apply to all plug-ins. Steps specific to particular plug-in formats will follow.

Special Gestalt The Auxiliary Output Stems API has a special gestalt associated with it that needs to be added: `pluginGestalt_SupportsAuxOutputStems`. Make sure this gestalt is added when defining types with multiple outputs. In the Sample Click example plug-in, this gestalt is added at the Type level in the Group class.

Adding Aux Output Stems in the Process Class The auxiliary outputs need to be defined at the initialization of the plug-in. To handle all the output stems and their info, a special manager has been created called the Auxiliary Output Stem (AOS) Manager. This manager is located in \TDMPlugIns\PlugInLibrary\Utilities\. It helps order and manage all the auxiliary outputs for every plug-in instance. The manager is essentially a vector that holds CAOSInfo objects; each CAOSInfo object holds all the necessary information for an auxiliary output.

To add an auxiliary output, first create a new `CAOSInfo` object for the output. The constructor for this object takes these as arguments:

- **stem format** either `ePlugIn_StemFormat_Mono` or `ePlugIn_StemFormat_Stereo`
- **port number** the number of the output, including all main outputs
- **aux output name** the name the plug-in gives the output

For example, to create a stereo auxiliary output stem object, code the following:

```
new CAOSInfo(ePlugIn_StemFormat_Stereo, portNumber, "myStereoAuxOutput1");
```

To aid in quickly generating aux output names, we have created a helper function, `AppendOutputPortNumToName(char *, int)`, that takes a string and appends an integer to it.

There are a few important things about the port number:

1. The port number of an aux output needs to be the lowest available port number after the main outputs of the track the plug-in is instantiated on. For example, the first available aux output for the plug-in residing on a 5.1 surround track would have a port number of 7, since there are 6 main outputs for the track. If the plug-in resided on a stereo track, the port number for this aux output would be 3, on a mono track it would be 2, so on and so forth.

2. As the AOS Manager is essentially a vector, port numbers must be declared sequentially and in the order aux output stems are added. For example, a stem can't be added with the port number 10 if it precedes a stem with the port number 4. The port number of the first stem must be one greater than the number of main outputs, the port number of the second stem one greater than that of the first stem, etc.

3. Stereo aux output stems will occupy two ports that must reside next to each other. Each stereo stem is referenced by its base port number, which is the port number of its left channel. Giving this base port number for a stereo stem implies that the right channel has a port number of one greater. For example, if a stereo stem has a base port number of 25, then it means that its left channel has the port number 25 and its right channel has the port number 26.

4. Each output channel must have a unique port number.

Once a `CAOSInfo` aux output stem object is created, you can add it by passing it into the `AddAuxOutputStem` function inherited from `CEffectProcess`.
The remaining steps to implementing Auxiliary Output Stems are RTAS- and MuSh- specific. This is because what remains is just to incorporate the extra outputs into the signal path, where a plug-in processes its audio. Auxiliary Output Stems is currently not supported for Non-MuSh TDM plug-ins.

RTAS-Specific Steps

The SDK enables you to implement RTAS in two different ways. The first way is to implement RTAS from an AudioSuite plug-in, using our AudioSuite classes. This is how RTAS plug-ins have been traditionally developed, even when an AudioSuite version is not released. The second way, which is much newer, is to implement RTAS alone without having to implement AudioSuite. Both methods utilize the Effect Layer, except the first method derives its process classes from `CEffectProcessAS` while the second method derives its process classes from `CEffectProcessRTAS`. A plug-in's audio processing occurs in `ProcessAudio` for traditional RTAS implementations and `RenderAudio` for the newer RTAS-only implementations.

For Auxiliary Output Stems, the plug-in just needs to account for the extra outputs when it processes the audio. Pro Tools will not automatically route your processed audio to all the extra outputs. As with main outputs, make sure the processed audio samples are placed in the auxiliary outputs' buffers as well. Additionally, if you are using `ProcessAudio`, only process output buffers that are connected. Unconnected outputs will be set to `NULL` in the `DAEConnectionPtr` buffer. Unconnected outputs will return `NULL` when you call `CEffectProcessAS::GetOutputConnection(long connectionIndex)`.

MuSh-Specific Steps

For MuSh plug-ins, all that remains to be done is to incorporate the extra outputs into the plug-in's internal audio path. The Effect Layer takes care of connecting and disconnecting your plug-in's aux outputs like it does with its main inputs and outputs. As with its main outputs, MuSh plug-ins need to push its outputted samples through these additional outputs in the DSP, which is executed in the DSP code. How this gets implemented is entirely up to the plug-in as each plug-in has different functionality and processes audio differently.

There is only one thing absolutely required when creating and declaring different effect types: `DefineNumAuxOutputPorts` must be called and give the number of aux output ports the plug-in will have. The syntax is such: `myAOSPlugInType->DefineNumAuxOutputPorts(N)`, where "N" is the number of aux output ports a plug-in has per instance.

Besides that requirement, make sure to correctly report the number of cycles needed per instance of the plug-in. Depending on how a plug-in incorporates Auxiliary Output Stems in its 56k DSP assembly code, this number may vary. For example, let's say a plug-in's the DSP code works so that it performs an additional "move" operation for each aux output. Then for every aux output it has, it will need to add an additional cycle to the cycle count it reports.

Feature: External Metering and Internal Clip

External Metering and Internal Clip are plug-in features that were introduced in Pro Tools 6.4. External Metering allows your plug-in meters to be accessed and delivered in real-time to an external control surface. Currently the only supported surface is D-Control, however future units may also provide support. Internal Clip is a new feature that is available to all users of Pro Tools. The plug-in header now has a clip light that indicates that the plug-in has clipped somewhere internally. Additionally, plug-ins that have clipped internally will appear in **red** on the insert, even if the plug-in window is not open. This allows users to see at a glance where clipping has occurred in their mix.

Effect Layer External Metering and Internal Clip

Adding External Metering and Internal Clip support to your Effect Layer plug-in is very straightforward, as long as you follow a few simple rules:

1. Add meters in your Process-level function `EffectInit`() via calls to `AddMeter()`. Make sure all meters are added and that they are added in the same order that you want them to appear horizontally across the top of the D-Control unit when it is in "Custom Fader" mode. Take a look at the Template sample plug-in `CTemplateProcess::EffectInit()` for an example. As you can see in the example, here you are given the opportunity to define properties of your meters that the D-Control unit will use: "type", "orientation", and "name" (currently not used).

2. Indicate that you support internal clipping by registering for the `pluginGestalt_SupportsClipMeter` gestalt at the Type level. See `CTemplateGroup`::`CreateEffectTypes`() for an example. Note that this gestalt is only required for indicating support for internal clipping. There is no gestalt required for indicating support for external metering.

3. Override the `CProcess` function `GetMetersFromDSPorRTAS`(). This function is used by the MeterManager to obtain real-time updates to your metering and clip data and **must** be implemented. Note that this function has a somewhat poorly chosen name. It should have been called something more along the lines of `PollForMeterAndClip()`, because it is in this function that you must retrieve all meter and clip information from the storage local to your host-based algorithm or from the DSP in the case of a TDM plug-in. This function is called regularly by the system via an external thread so that metering and clip information can be obtained even when the plug-in UI is closed. If you poll for metering and/or clip data during `DoTokenIdle()` you will have problems, as this function is only called when the plug-in GUI is open.

   Example implementations of this function can be found in `CTemplateProcessTDM.cpp`, `CTemplateProcessMuSh.cpp`, `CTemplateProcessAS.cpp, and CTemplateProcessRTAS.cpp`. In addition, you must follow these rules in your implementation of this function:

   a. Meters must be returned in the same order as you established via your calls to `AddMeter`() in `EffectInit`().

   b. If clipping occurs you must return a value of –1 for the meter. You must also set the `fClipped` variable (defined in the `CProcess` base class) to true. This will ensure that the clip light in the plug-in header is properly lit when a clip occurs. This is true even if you detect a "true" internal clip that is not associated with any meter (and is therefore not reported back in the clipIndicators array).

   c. Return the value of the metering data in 32-bit integer form. This is the format that the D-Control personality expects. This is true even if you are reporting metering data for something like a gain reduction meter, which might normally be reported in decibels. Here you must convert from decibels to "gain multiplier" in 32-bit integer (see discussion below).

   d. This function is called from a secondary thread, and so for Non-MuSh TDM plug-ins you must wrap any DSP access sequences with calls to `Lock()` and `Unlock()`. See Tech Note #13.

4. When doing your drawing of meters or clip lights in your GUI you must get the meter values from the MeterManager via a call to `GetMetersAndClipIndicators`(). DO NOT call `GetMetersFromDSPorRTAS`() directly! The MeterManager contains logic for throttling the number of metering related calls that occur and also for managing metering data so that

peaks are not missed either in your GUI or on the D-Control unit. `GetMetersAndClipIndicators`() will in turn lead to a call to your `GetMetersFromDSPorRTAS`() function. See `CTemplateProcess::UpdateMeters`() for an example.

Optionally, if you add additional clip lights to your plug-in GUI (as in the Template sample plug-in)...

5. Override the virtual function `CProcess::ClearClipState`(). This function is called when the user clicks the clip indicator in the plug-in header. You should clear all your internal clip lights when this occurs. See `CTemplateProcess::ClearClipState`() for an example.

6. Add logic so that if ALL of your internal clip lights are cleared by the user, then the `fClipped` variable is set to false. This is important because it allows the clip indicator in the plug-in header to be cleared when all internal clip lights are cleared. We have provided a new view class in the 6.4 SDK called `CClippingView` that implements an internal clip light UI element, which you may find useful. Take a look at `CTemplateProcess::SetClipped`() to see where this occurs. `SetClipped`() is a function that is called by the `CClippingView` object when it is clicked.

Optionally, if your algorithm is such that clipping can occur internal to the algorithm (i.e. not on either an input or output meter, for example)...

7. In this case there are no meters associated with the clip. You should simply set the `fClipped` variable to TRUE in `GetMetersFromDSPorRTAS()` when you detect internal clipping like this. This will allow the clip indicator in the plug-in header to light up properly. Some plug-ins may wish to provide a separate UI element that indicates such internal clipping. Reverb One and ReVibe are examples of Digidesign plug-ins that have a separate internal clip light like this.

Testing with the Emulator

Once you have completed your external metering and internal clip implementation, you should test this implementation using the D-Control emulator. The emulator will show live metering and will show clip indication. To see your plug-in meters in action in the emulator you must be in “Custom Fader” mode. See the emulator documentation for more details.

You may notice some unexpected behavior when using the emulator:

1. Clearing all your internal clip lights will cause the clip indicator in the plug-in header to clear, but not the clip indicators in the D-Control emulator. The user must click on the plug-in header clip light to clear the emulator.

2. Clearing an individual clip light in the plug-in UI will not clear the corresponding individual clip indicator in the emulator. Experiment with the Stereo or 5.1 Template plug-in to see an example.

Additional Considerations

Meter Types and Orientations

`FicPlugInEnums.h` defines the available meter types and their orientations. Meter types are important, because they are used to map a plug-in’s meters to the appropriate meters on the D-Control’s center Dynamics and EQ sections. If your plug-in supports the center section by defining the

appropriate page tables (see the **Control Surfaces and Page Tables** chapter), then its meters will be displayed on this section using the following rules: all meters of type `meterType_Input` will be summed, averaged and then displayed in the Input meter in the center section. Likewise for meters of type `meterType_Output` on the Output meter, meters of type `meterType_CLGain` on the Compressor/Limiter meter, and meters of type `meterType_EGGain` on the Expander/Gate meter. Note that `meterType_Analysis` and `meterType_Other` currently are not implemented and should not be used.

Meter orientations define how the meter will be drawn on the control surface when in "Custom Fader" mode. Meters with `meterOrientation_BottomLeft` will be drawn from the bottom up (there are no horizontal meters on the D-Control). `meterOrientation_TopRight` meters will be drawn from the top down. `meterOrientation_Center` and meterOrientation_PhaseDot are only used on the D-Control control surface.

Note that the meter orientation that you define will have no effect in the center Dynamics and EQ sections. In these sections how meters are drawn is dependent on the `EMeterType` that you define for the meter. All input and output meters will be drawn from bottom up and all Compressor/Limiter and Expander/Gate meters will be drawn from the top down.

Metering Data Format

As mentioned, in order for meters to show up correctly on the control surface they must be reported in a very specific format: "gain multiplier" in signed 32-bit integer. What do we mean "gain multiplier"? By this we mean the ratio "X" that appears in the standard decibel formula*:

$dB = 20 * \log_{10}X$, where X = (sample level / full scale)

or, solved for X:

$X = 10^{(dB/20)}$

To express this ratio as a signed 32-bit integer we must multiply the ratio by the largest number representable in a signed 32-bit integer: 0x7FFFFFFF.

So, putting all this together, if your meters are expressed in dB, you must convert to the correct format using the following formula:

$10^{(dB/20)} * 0x7FFFFFFF$

*Note that this assumes dBFS.

The D-Control personality will take this metering data and apply the reverse formula on it to obtain the dB representation. It will then map the value to the 32-segment LED meter using the following look-up table:

Value 0 = 2145012687 --- -0.010000 dBs
Value 1 = 2027355293 --- -0.500000 dBs
Value 2 = 1913946815 --- -1.000000 dBs
Value 3 = 1765752644 --- -1.700000 dBs
Value 4 = 1629032937 --- -2.400000 dBs
Value 5 = 1502899241 --- -3.100000 dBs
Value 6 = 1386531898 --- -3.800000 dBs
Value 7 = 1279174712 --- -4.500000 dBs
Value 8 = 1180130039 --- -5.200000 dBs
Value 9 = 1076291388 --- -6.000000 dBs
Value 10 = 981589412 --- -6.800000 dBs

Value 11 = 895220183 --- -7.600000 dBs
Value 12 = 816450511 --- -8.400000 dBs
Value 13 = 744611716 --- -9.200000 dBs
Value 14 = 679093956 --- -10.000000 dBs
Value 15 = 605243125 --- -11.000000 dBs
Value 16 = 539423503 --- -12.000000 dBs
Value 17 = 480761703 --- -13.000000 dBs
Value 18 = 404510561 --- -14.500000 dBs
Value 19 = 340353221 --- -16.000000 dBs
Value 20 = 286371546 --- -17.500000 dBs
Value 21 = 240951628 --- -19.000000 dBs
Value 22 = 202735529 --- -20.500000 dBs
Value 23 = 170580689 --- -22.000000 dBs
Value 24 = 135497057 --- -24.000000 dBs
Value 25 = 107629138 --- -26.000000 dBs
Value 26 = 76195595 --- -29.000000 dBs
Value 27 = 53942350 --- -32.000000 dBs
Value 28 = 34035322 --- -36.000000 dBs
Value 29 = 17058068 --- -42.000000 dBs
Value 30 = 6790939 --- -50.000000 dBs
Value 31 = 2147483 --- -60.000000 dBs

We provide this information in the event that you wish to represent meters using something other than dB, for example a linear meter of some type. In the future we hope to provide direct support for linear and other types of meters, but in the meantime you'll need to "cook" the metering data before sending it to the control surface personality if you wish to do this.

Non-Effect Layer External Metering and Internal Clip

Adding External Metering and Internal Clip support to a non-Effect Layer plug-in requires considerably more work. Here we list the functions that must be overridden to implement support for this feature in a non-EL plug-in:

```
ComponentResult CProcess::GetNumMeters(long* aNumMeters)
```

Returns the total number of meters that the plug-in supports. i.e. the sum of all input/gain reduction/output/etc. meters that a plug-in might support.

```
ComponentResult CProcess::GetMeterVal (long aMeterIndex, long* aValue)
```

Based upon `GetNumMeters`() is a one-based metering index. A 32 bit signed integer is returned, 0x00000000 (minimum) to 0x7FFFFFFF (maximum), any number less than zero indicates the meter has clipped.

```
ComponentResult CProcess::GetMeterName (long aMeterIndex, char* aNameString)
```

Returns a `CString`. (Please Note: this routine was created to help with the debugging of the mixer plug-in. Consider implementation as "optional", until further notice.)

```
ComponentResult CProcess::GetMeterOrientation(long aMeterIndex,
EMeterOrientation* anOrientation)
```

Returns a 16 bit integer corresponding to the orientation of the meter. The current list of `eMeterOrientation` enums (also found in `FicPluginEnums.h`) is:

`meterOrientation_BottomLeft` // the default orientation

```
meterOrientation_TopRight  // Some dynamics plug-ins orient their gain reduction like so
meterOrientation_Center     // A plug-in that does gain increase and decrease may want this
                            // meter values less than 0.5 would display downward from the mid-point
                            // meter values greater than 0.5 would display upward from the mid-point
```

```
ComponentResult CProcess::GetNumMetersOfType(EMeterType   meterTypeSelector,
long* aNumMeters)
```

Returns the total number of meters that match the type selector. The current list of `eMeterType` enums (also found in `FicPluginEnums.h`) is:

```
meterType_Input             // e.g. Your typical input meter (possibly after an input gain stage)
meterType_Output            // e.g. Your typical output meter (possibly after an output gain stage)
meterType_CLGain            // e.g. Compressor/Limiter gain reduction
meterType_EGGain            // e.g. Expander/Gate gain reduction
meterType_Analysis          // e.g. multi-band amplitude from a Spectrum analyzer
meterType_Other             // e.g. a meter that does not fit in any of the above categories
```

```
ComponentResult CProcess::GetMeterIndexByType(EMeterType meterTypeSelector, long
typeIndex, long* aMeterIndex)
```

Return a one based meter index that can be used by `GetMeterVal`() or `GetMeterOrientation`(). `typeIndex` is a sub-index that ranges from one to `GetNumMetersOfType`(), for a given `meterTypeSelector`. As an example, this routine could be used by Buckley when it needs to display plug-in output on a single specific meter.

The Host Application can first call `GetNumMetersOfType(eMeterType_Output, numMeters)` to find out how many output meters the plug-in has (let's assume more than one for the sake of this example), and then iterate through that subset of meters, getting each meters proper `meterIndex`, so that the maximum, aggregated meter value can be computed.

Internal Clip Indication

At the CProcessType level the Gestalt method will have a new selector: `pluginGestalt_SupportsClipMeter`. Plug-ins that implement the following two routines (listed below) should return "true" for `pluginGestalt_SupportsClipMeter`. Non-compliant plug-ins will, by default, return "false" for either selector that is not implemented.

```
ComponentResult CProcess::GetClipState(bool* isClipped)
```

This routine returns true or false. "True" indicates some form of clipping (either at an input gain stage, or during internal signal processing, or at the output gain stage, or at a summing node, somewhere, anywhere) has occurred since the last time `GetClipState` was called.

It is up to the plug-in to determine how thorough its clip detection should be. Ideally every plug-in would detect any clip that occurs anywhere during its signal path. In reality, there is often and tradeoff between the thoroughness of the clip detection, and the efficiency of the DSP code. While some plug-ins may test the Saturation bit on the DSP for any trace of a clip, others may simply leverage off of preexisting output metering. In any case, some form of clip detection needs to occur.

```
ComponentResult CProcess::ClearClipState()
```

This routine is called to tell a plug-in that it should clear any clip indicators that it may own within its plug-in GUI. Note: Pro Tools will have a generic clip light in the top portion of the plug-in window. This light will be turned on and off by Pro Tools based upon `GetClipState`(). This means most plug-ins will

not need to override `ClearClipState`(). It is only plug-ins that have "sticky" clip indicators within their own GUI that will need to clear their clip lights when `ClearClipState` is called.

One important consideration: With the advent of plug-in metering simultaneously in more than one location (control surface and computer GUI) it is important that neither meter "hide" meter data from the other. Since most DSP implementations clear their metering accumulator after returning the most recent metering max, it is important to share that metering data between multiple meters. Otherwise, if the GUI meter got an output meter value from the DSP, and soon thereafter an external meter got another output meter value, then the external meter would likely show an inaccurate meter level because the GUI meter value wasn't shared and the DSP might have been in a low waveform section during its very short time interval.

The Meter Manager Class

With this in mind, a new class has been implemented called `CMeterManager`, which manages the metering data by saving the meter values and clipping states for a given plug-in. The meter manager will then pass the data out instead of the currently retrieved value in the case that the caller to the meter manager is different from the previous caller, and the saved meter data is larger than the batch that has just been retrieved. For instance, if the internal plug-in GUI makes a call to the meter manager and gets a value of 0x6fffffff for the left output meter, before the meter manager will let this value out for reporting it will check to see if the previous call was also by the internal GUI or by some external(i.e. Procontrol) source. If the previous call was by an external source, the meter manager will then check to see if its stored value is larger than the 0x6fffffff that was just recieved. If it is larger, then this larger value will be reported out. If it is not larger, then the manager will store the 0x6fffffff into its left output meter value, and store the new previous caller as the internal GUI for the next time around. On the other hand, if the previous caller was the internal GUI (two consecutive meter calls by the same source) then the value we just recieved (0x6fffffff) is stored in the meter manager and reported out to the caller as usual.

The meter manager also has a provision for getting the meter values and then simply dumping them and reporting zeros back to the caller in order to clear any stale values. This is especially useful when the GUI is closed on screen, but the plug-in becomes clipped due to signal running through it. In this situation, the plug-in is not checking for meter values because it is closed, and the external `GetClipState` will stop checking until a clear clip message comes through. What winds up occuring in this situation is that a clear clip comes through from the user, and the `GetClipState` will begin checking to see if the meters indicate clipping again. Unfortunately, the first value that will be returned is the stale full scale value that was last stored in the plug-in's DSP or RTAS meter cache. This clear value provision allows a call to get rid of any such stale values.

All of the action in this class takes place in the `AccessMeters` function.

```
void AccessMeters (char caller, long *allMeters, bool *clipIndicators, long
meterIndex = -1, long *meterValue = 0, bool clearValuesOnly = false)
```

As explained above, this function takes a character to indicate who the call is coming from, (i.e. 'e' for external or 'i' for internal), an array of longs, and an array of bools. The function can also take a meter index and a pointer to a long if the caller is only interested in a single value. Finally, there is the clear boolean that can be set to dump meter values as mentioned above.

To use the meter manager, a function will need to be written in your plug-in called `GetMetersFromDSPorRTAS`.

```
virtual void MyPlugIn::GetMetersFromDSPorRTAS(long *allMeters, bool
*clipIndicators)
```

This function takes in an array of longs and an array of bools and simply returns all of the meter values for the plug-in in the long array, with the bool array being set to true for any meters that have indicated an over.

What the meter manager's AccessMeters function and the associated GetMetersFromDSPorRTAS function do is eliminate most of what the "UpdateMeters" function does in a plug-in. Whereas the UpdateMeters was responsible for getting all of the meter values, scaling them, and then printing them to the GUI, now the UpdateMeters simply becomes:

```
void CMyPlugInProcess::UpdateMeters (void)

{
       mMeterManager->AccessMeters('i', mMeterValues, mClipValues);
       this->PrintMetersToScreen(mMeterValues, mClipValues);
}
```

Setting Up The Meter Manager

To use the meter manager, simply include it in the header for your process, (CMeterManager is in the PlugIn library under the view classes.) In your process's header, define the manager as follows:

```
CMeterManager *mMeterManager;
```

Then in your initialization routine, declare the meter manager with a pointer to your process and the total number of meters that you will need to keep track of:

```
GetNumAudioOutputs(&mNumOutputs);

GetNumAudioInputs(&mNumInputs);

mMeterManager = new CMeterManager(this, mNumInputs + mNumOutputs + 1); // input
meters, output meter, and one gain reduction meter
```

Feature: Automatic Delay Compensation

Pro Tools 6.4 TDM introduces Automatic Delay Compensation for maintaining time-alignment between tracks that have plug-ins with differing DSP delays, tracks with different mixing paths, tracks that are split off and recombined within the mixer, and tracks with hardware inserts. To maintain time alignment, Pro Tools adds the exact amount of delay to each track necessary to make that particular track's delay equal to the delay of the track that has the longest delay. This feature is only available in TDM versions of Pro Tools, and not in Pro Tools LE.

Because of this new feature, it is now imperative that plug-in sample delays get reported correctly to DAE. Plug-ins should indicate the amount of delay by overriding the CProcess function GetDelaySamplesLong(). How many samples of delay to report differs depending on whether the plug-in is TDM, MuSh, or RTAS:

1. TDM Plug-ins

   For a TDM plug-in, the value reported should only account for the delay caused by the .asm code and not TDM delays. For example, if within a plug-in the incoming samples are read, processed, then output on the same TDM interrupt, then the delay would be zero. If the samples are double-buffered, as in the case of the Template non-MuSh sample plug-in, the delay would be one sample.

2. MultiShell Plug-ins

   MuSh plug-ins should follow the same rules as for TDM plug-ins, except that they should add 2 samples of delay to account for the standard amount of MultiShell delay. For example, in the MuSh target of the Template sample plug-in, the .asm code does not follow

the double-buffering scheme so we report 2 samples of delay in `GetDelaySamplesLong()` – 2 for MuSh plus 0 for our algorithm.

3. RTAS Plug-ins

   While disk track slipping of the number of samples in the RTAS callback buffer has been around for a while to compensate for RTAS latency, true ADC for RTAS plug-ins only arrived with Pro Tools 6.4. Because disk track slipping of the callback buffer is already assumed, an RTAS plug-in should report 0 samples of delay unless it processes on blocks of audio larger than the size of this buffer. If the plug-in must buffer more samples than the size of one RTAS callback buffer before producing any output, it should return the exact number of samples the output is delayed. Please note that delay compensation for RTAS plug-ins occurs on the TDM buss, and does not employ disk track slipping.

A note regarding ADC and Bypassing: It is recommended that your plug-in incur the same amount of delay when it is bypassed as when it is not, so that should a user manually compensate for the delay, their audio does not become out of phase when the plug-in is bypassed.

Feature: Plug-In Categories

Beginning in Pro Tools 6.4, the user has the option of making the plug-in menus hierarchical. That is, if the user chooses, plug-ins can be sorted in the Insert and AudioSuite menus according to the categories listed below. Plug-ins have had the ability to report a category for a while now, but with the addition of the hierarchical plug-in menu feature in Pro Tools, it is now critical that plug-ins report the proper category.

A plug-in category is a general description of the signal processing function of a plug-in. The category can be used in cases where the application requires knowledge of the plug-in type, and based on that category, can apply special treatment.

These general categories represent common plug-in functions like EQ, dynamics, reverb, etc. See `FicPluginEnums.h` for the most recent set of constants, named EPlugInCategory. As of this writing, the following categories are defined with descriptions of the types of plug-ins that would fit in the category:

`ePlugInCategory_None`	**(plug-ins with no category will be placed in the "Other" menu)**
`ePlugInCategory_EQ`	**equalization**
`ePlugInCategory_Dynamics`	**compressor, expander, limiter, gate, etc.**
`ePlugInCategory_PitchShift`	**pitch processing**
`ePlugInCategory_Reverb`	**reverberation, room simulation**
`ePlugInCategory_Delay`	**delay, multi-tap delay**
`ePlugInCategory_Modulation`	**phasing, flanging, chorus, etc.**
`ePlugInCategory_Harmonic`	**distortion, tape simulation, sub-harmonic**
`ePlugInCategory_NoiseReduction`	**noise reduction**
`ePlugInCategory_Dither`	**dither, noise shaping, UV22, etc.**
`ePlugInCategory_SoundField`	**pan, auto-pan, Dolby 5.1, surround effects not fitting in another category**
`ePlugInCategory_HWGenerators`	**fixed hardware sources such as SampleCell**
`ePlugInCategory_SWGenerators`	**software based synths, samplers, metronomes, running on generic DSPs, including VST to RTAS wrapped synths**
`ePlugInCategory_WrappedPlugin`	**all plugins wrapped by a thrid party wrapper (i.e. VST to RTAS wrapper), except for VST synth plugins which should be mapped to ePlugInCategory_SWGenerators**
`ePlugInCategory_Effect`	**multi-effect plug-ins and effect plug-ins that do not fit in other categories**

Note: There is still room for defining additional categories if you have a plug-in that doesn't easily fall into one of the types listed above. If you think you have a category that may prove useful, please contact developer services so that it can be considered for inclusion. Email: devservices@digidesign.com.

What To Do

If you are utilizing the Effect Layer, the category is simply defined within the constructor of the Effect Type.

Otherwise, your plug-in reports its category via the `CYourProcessType::Gestalt()` method. A new selector called `pluginGestalt_CategoryBits` has been added for this purpose. For example, a limiter plug-in would add the following:

```
case pluginGestalt_CategoryBits:
   *aResultP = ePlugInCategory_Dynamics;
   break;
```

It is also possible for a plug-in to have multiple categories, since CategoryBits is a bitmask. If yours does, it will require additional work. Please see the "Plug-Ins With Multiple Categories" section in the **Control Surfaces and Page Tables** chapter for a discussion.

Feature: Saving and Restoring Plug-In Settings

Pro Tools can save and restore plug-in settings data into either a Pro Tools settings file (.tfx) or a user-specified file type. The following section describes the latter. For more information on how plug-in settings are saved and restored see "Saving & Restoring Custom Data" in the **Using the Effect Layer** chapter and "Saving And Restoring Settings (Without using the Effect Layer)" in the **General Topics** chapter.

Saving Files of Any Type with a Pro Tools Session

In Pro Tools 7.0, a new API was added that allows plug-ins to store files of any format with a Pro Tools session. DoCustomPlugInSettingsFile() returns the path to a plug-in's session settings file directory. This information allows plug-ins to save files with a session, even if they are not .tfx format. In order to take advantage of this, plug-ins must register one plug-in gestalt, and override one function in the plug-in process class:

pluginGestalt_UsesCustomPlugInSettingsFile - This gestalt must be defined at the type level.

DoCustomPlugInSettingsFile() - This function must be overridden in a plug-in's process class. It is called at plug-in instantiation, before any calls to GetChunk() or SetChunk(). This allows the plug-in to save information about the plug-in session settings folder into the preset automatically saved by Pro Tools, using a custom chunk.

Files placed in the plug-in session settings folder will be copied when a user selects "Save Session Copy In..." from the Pro Tools File menu, provided the user chooses to copy plug-in settings files. However, if a track containing the plug-in is imported into a new session using the "Import Session Data..." command, the custom files saved with the original session will not be copied to the new session.

Manually Loading .tfx Settings Files

There is a new API in Pro Tools 7.3 that will allow a plug-in to manually load a .tfx plug-settings file. The plug-in can pass Pro Tools a path for a .tfx file, and Pro Tools will load via the SetChunk function, just as if the user had loaded the file. To take advantage of feature, a plug-in must define `pluginGestalt_WantsLoadSettingsFileCallback`, override `SpecifyLoadPlugInSettingsFileCallback()` in its Process class. Pro Tools call into this function after instantiation and pass the plug-in a

`LoadPlugInSettingsFileCallback` function pointer, defined in TDMFicEngine.h. The plug-in can then use this callback into Pro Tools to request a .tfx file to be loaded.

Feature: Drag -N- Drop Functionality

Since version 7.0, Pro Tools has offered Drag-N-Drop (DND) functionality for plug-ins. Developers can allow the user to Drag-N-Drop certain file types from the Pro Tools Workspace into their plug-in. Currently only audio files are supported by DND, but support for other file types may be added in the future. Two plug-in gestalts are defined for DND, one to let DAE know that the plug-in supports DND and another to allow for custom highlighting for DND. There are five process level functions calls that will need to be overridden by the plug-in to implement DND functionality. Complete documentation is available in the comments of CProcess.h and the doxygen auto-generated documentation available on developer.digidesign.com. See below for an overview of the specific gestalts and functions relevant to DND.

Drag And Drop Gestalts:

`plugInGestalt_SupportsDND`, // 1 = is able to handle reception of Drag and Drop file data, 0 = doesn't support DND file drop

`plugInGestalt_SupportsCustomDNDHilight`, // 1 = want to do its own hilighting during Drag And Drop, 0 = will use default hilight Drag And Drop Methods:

Drag And Drop Functions:

```
virtual ComponentResult DrawDragAndDropHilight (long horizontalPosition,
                                                long verticalPosition,
                                                bool turnHilightOn);
```

OVERRIDE: Called by host application. Informs a plug-in of where to draw "custom" drag and drop highlighting.

This method is invoked by DAE to allow a plug-in to draw (or erase) visual feedback during the drag phase of a drag and drop. This method will only get called if the `plugInGestalt_SupportsDND` and `plugInGestalt_SupportsCustomDNDHilight` gestalts are supported. If `plugInGestalt_SupportsCustomDNDHilight` is not supported, default highlight drawing (i.e. a framing reactangle) will be done automatically for the plug-in.

Parameter: horizontalPosition is the horizontal mouse position localized to the plug-in view.
Parameter: verticalPosition is the vertical mouse position localized to the plug-in view.
Parameter: turnHilightOn informs the plug-in on whether it is time to erase (turnHilightOn == false) the previously drawn position, or draw (turnHilightOn == true) the new position

```
virtual ComponentResult BeginDragData(bool* useFSSpecs);
```

OVERRIDE: Called by host application. Informs a plug-in that a sequence of RegisterDragData calls is about to begin.

This method is invoked by DAE to allow a plug-in to prepare to receive drag data. A plug-in may want to create an empty list to store forthcoming drag objects.

IMPORTANT NOTE: The `plugInGestalt_SupportsDND` gestalt must be supported to enable this and the other drag and drop methods.

Parameter: useFSSpecs is a return value that allows a plug-in to specify whether file object references should be delivered as Mac native FSSpec pointers (useFSSpecs == true) or Windows native Path String pointers (useFSSpecs == false).

```
virtual ComponentResult RegisterDragData( long       dataFlavor,
                                          void*objectRefPtr,
                                          bool*      accepted);
```

OVERRIDE: Called by host application. Describes a drag object to a plug-in.

This method is invoked by DAE to allow a plug-in to make note of an object that is being dragged. The plug-in can report back as to whether the object is acceptable or not. If no objects are acceptable to the plug-in, no objects will be dropped on the plug-in.

IMPORTANT NOTE: Data referenced by "objectRefPtr" should be copied to local storage. Host based data structures may be reused upon subsequent calls to "RegisterDragData".

Parameter: dataFlavor indicates the type of data that is being dragged (e.g. eDragAndDrop_AudioFile)
Parameter: objectRefPtr is a reference to the drag and drop data. If dataflavor is eDragAndDrop_AudioFile, a file reference will be returned that is either a pointer to an FSSpec or a pointer to a file path string.
Parameter: accepted informs the host application on whether the current drag object is acceptable by the plug-in.

Currently supported dataFlavors:
eDragAndDrop_AudioFile = 0 // In the future more drag and drop data flavors may be supported

```
virtual ComponentResult CompleteDragData(void);
```

OVERRIDE: Called by host application. Informs a plug-in that a sequence of RegisterDragData calls is completed.

This method is invoked by DAE to allow a plug-in to do any post processing it might need to do.

IMPORTANT NOTE: Although a set of drag objects may have been presented to a plug-in. The objects have not been dropped yet. It is possible that the user dragged across the plug-in window en route to another drop target. A plug-in must wait until "DropData" data is called to determine if the drop occurred in the plug-in.

```
virtual ComponentResult DropData(long   horizontalPosition,
                                 long   verticalPosition);
```

OVERRIDE: Called by host application. Informs a plug-in on whether the data drop occurred within the plug-in.

This method is invoked by DAE to inform a plug-in of the drop location.

IMPORTANT NOTE: It is possible that the user dragged across the plug-in window en route to another drop target. In this case, negative drop coordinates indicate that the drop did not occur in this plug-in's UI.

Parameter: horizontalPosition is the horizontal mouse position localized to the plug-in view where the drop occured. A negative coordinate indicates that the drop occurred outside the plug-in window.
Parameter: verticalPosition is the vertical mouse position localized to the plug-in view. A negative coordinate indicates that the drop occurred outside the plug-in window.

Drag And Drop Type Additions:

In Pro Tools 7.3, new data types can be sent to plug-ins that support drag and drop:
The first is `eDragAndDrop_PTAudioRegion`, which represents a region from the timeline or the region bin of the Pro Tools Edit Window. If a plug-in supports this new type, it will receive a `SPTAudioRegion` object, defined in TDMFicEngine.h, for each Pro Tools region.
The other new type is `eDragAndDrop_UnknownDragData`, which represents a file of unknown type. A user can now drag any file onto a plug-in window from the Pro Tools workspace, and if the plug-in supports the `eDragAndDrop_UnknownDragData` type, it will receive a path to this file.

Additionally, it is possible for a user to drag a .tfx plug-in settings file onto a plug-in, and Pro Tools will load that settings file via the `SetChunk` call in the plug-in. This is handled automatically by Pro Tools for plug-ins that utilize the Digidesign View classes. If your plug-in does not use the View classes, please see the section in this document titled “Drag and Drop Now Supported for Template_NoUI Based Plug-Ins“ for information on how to implement drag and drop, and the section titled “Feature: Saving and Restoring Plug-In Settings“ to load a .tfx file once it has been dropped onto the plug-in window.

Support for Template_NoUI-based Plug-Ins:

Plug-ins based on the Template_NoUI sample plug-in (described in Chapter 17), and therefore do not use the plug-in SDK View widgets, can support drag and drop in Pro Tools 7.3. If a plug-in has defined `pluginGestalt_DoesNotUseDigiUI`, Pro Tools will send standard OS drag and drop events to the plug-in window. Two flavors of data can be sent to a plug-in. A .tfx plug-in settings file will have the data flavor “Digidesign Plug-In Settings File data” on Windows, and ‘PISF’ on Mac. Any other file will have the data flavor “Digidesign Browser File data” on Windows, and ‘DGBF’ on Mac. In either case, the data will be a struct of type `OSStyleDragNDropFileList`, defined in TDMFicEngine.h.
Pro Tools audio regions will not be delivered using either of these two data flavors. They will be delivered via the existing Drag and Drop implementation for both “NoUI” plug-ins and plug-ins that use the View classes.

Feature: Plug-In Automation

Plug-in controls can be enabled to use the automation system in Pro Tools. The automation is achieved by using the SDS Token System, which is described in greater detail in the section: “SDS Token

Automation System" in the Graphical User Interface chapter. (Note: It is important to know that the token system is asynchronous and not sample accurate.) All controls subclassed from either CPlugInControl_Continuous or CPlugInControl_Discrete take the parameter: **`bool isAutomatable`** on construction. Specifying this parameter **`true`** allows a control to be enabled for automation, whereas **`false`** prohibits the control from being automated.

Feature: Sidechain Inputs

If applicable, plug-ins may choose to enable sidechain inputs. If a sidechain is enabled, a menu is added to the plug-in's header that allows the user to choose an interface or bus as the sidechain, or "key input". For AudioSuite, the user can only use an existing audio track as the sidechain input. The following plug-in gestalt allows each type to enable sidechain input:

`pluginGestalt_SideChainInput`

Once enabled, the plug-in will be able to access sidechain input just like any other input signal. For AudioSuite and RTAS types, the sidechain buffer is always the last input channel. Call **`CEffectProcess::GetSideChainConnectionNum()`** to get the channel number. For more details on this function see the Effect Layer Reference section. For information on accessing sidechain audio from a TDM type, please see the chapter **Writing TDM DSP Code**.

Feature: Plug-In Streaming Manager

In Pro Tools 7.3, the Plug-in Streaming Manager (PSM) API was introduced to allow plug-ins an easy method of streaming data from disk. The PSM is an asynchronous cross-platform disk I/O interface that uses the Pro Tools disk scheduler. When used correctly, the PSM makes it possible to share audio volumes for both tracking and file streaming without suffering the poor performance typically associated with such a scenario. This new API is documented in Tech Note 23: "Plug-In Streaming Manager: Streaming Data from Disk", which is available at: http://developer.digidesign.com -> Tech Notes.

Feature: Pro Tools Session Browser Integration

Plug-ins can now integrate with the Pro Tools Session Browser tool, providing a more seamless user experience for plug-ins that rely on external files. A plug-in can notify Pro Tools of dependencies on external files, and these files will be handled as audio file dependencies in the session. They will appear in the Session Browser, users will have an option to copy the files with the session during a "Save Session Copy In..." or "Import Session Data..." operation, and if any files are missing when opening an existing session, the user will be prompted to find them via the "Find and Relink" dialogue that is used for missing audio tracks and region data.

Note: This feature is currently intended for plug-in sample data. Although it is possible for plug-ins to set any file as a dependency in the Session Browser, all dependencies are labeled as "Sampler Files" in the Session Browser.

To implement this feature, a plug-in must define `pluginGestalt_WantsFindAndRelink` and implement three functions at the Process level: `SpecifyFindAndRelinkCallback()`, `ReturnFindAndRelinkFiles()`, and `SetFilePathsForSessionCopy()`. These functions are documented in more depth in CProcess.h. The procedure for notifying Pro Tools of audio file dependencies is as follows:

1) When a plug-in is instantiated, Pro Tools will check to see if it supports `pluginGestalt_WantsFindAndRelink`, and if so, Pro Tools will call `SpecifyFindAndRelinkCallback()` at the Process level. `SpecifyFindAndRelinkCallback()` passes a function pointer of type `FindAndRelinkCallbackPtr` to the plug-in. `FindAndRelinkCallbackPtr` is documented in TDMFicEngine.h.

2) The plug-in calls into Pro Tools via the `FindAndRelinkCallbackPtr` function pointer, and notifies Pro Tools of any plug-in file dependencies. The plug-in can pass three file lists: 1) files which need to be added to the plug-in's dependency list, 2) files which need to be removed from the plug-in's dependency list, and 3) files which need to be added to the plug-in's dependency list, but which the plug-in cannot find on disk. Pro Tools will use the "Find and Relink" functionality to prompt the user to locate any files in this third list. Pro Tools will return immediately from this callback.

3) Once the "Find and Relink" process is finished, Pro Tools will call `ReturnFindAndRelinkFiles()` at the Process level and deliver new file paths to the plug-in for any files found by the user. If the plug-in did not request that Pro Tools "Find and Relink" any files, or if none of the requested files were found by the user, `ReturnFindAndRelinkFiles()` will not be called.

4) If the file dependencies change at a later time (for example after a patch change in a sampler), the plug-in can repeat steps 2 and 3 to add or remove files from the dependency list.

When opening a previously saved session, a plug-in's saved settings may have existing file dependencies. The plug-in will need to call into the `FindAndRelinkCallBackPtr` when it is instantiated, in order to notify Pro Tools of the file dependencies for the saved patch. For this reason, the plug-in should save a list of all file dependencies in its settings file, by using a custom chunk.

However, when doing a "Save Session Copy In..." operation, the user may choose to copy a plug-in's file dependencies to the new location of the session copy. In this case, the file paths saved into the plug-in's custom chunk would be incorrect in the new session. In order to accurately save plug-in settings during a session copy, the plug-in should implement `SetFilePathsForSessionCopy()` as described below:

1) When a user initiates a "Save Session Copy In..." operation, Pro Tools will call `SetFilePathsForSessionCopy()` at the plug-in's Process level. Pro Tools will pass two lists, a list of the current file dependencies registered by the plug-in, and a list of the new locations of each of these files after the session copy. The plug-in should save the list of new locations, so they can be written into the plug-in settings for the session copy. Pro Tools will also set the `Boolean isSessionCopy` parameter to true.

2) Pro Tools will call `GetChunk()` in the plug-in to write out the settings for the new session file. The plug-in should use the list of new locations delivered in step 1 for its custom chunk.

3) Pro Tools will call `SetFilePathsForSessionCopy()` again with `isSessionCopy` set to false. All subsequent `GetChunk()` calls will be for the current session, not the session copy. The plug-in no longer needs the list of new locations delivered in step 1, and can go back to writing the current locations into its custom chunk.

Part II: The User Interface

Chapter 5: The Graphical User Interface

The legacy of the plug-in Library is strongly rooted in the MacOS. This is especially true for elements of the graphical user interface. To develop plug-ins that run on both the MacOS and Windows platform, we rely on the Mac2Win™ Porting Technology by Altura Software, Inc. This solution has allowed us to maintain our code base for the plug-in Library and allows you to develop a single plug-in that is simultaneously compatible on both platforms.

Key Concepts

The process of developing a GUI for a DAE plug-in has sometimes been described as confusing. To an extent, this is true for those approaching the architecture for the first time. However, by initially illuminating a few key points , giving you some historical background along the way, and frankly presenting some of the limitations of the process, hopefully a clear and easy picture of the GUI system will be presented.

First off, those key points:

◆ The physical layout of the plug-in window is defined within the Resource Fork of the plug-in. For the Windows platform, this fork data is extracted and placed in a separate file, using the provided `swap_resource_to_data.sh` script. (Mac2Win™ handles the run-time translation of accessing this file instead of the non-existent Resource Fork.) Typically, the Macintosh program `ResEdit` is utilized to create and edit these Resources.

◆ At run-time, upon instantiation, the class `CDialogView` parses the Resource information and generates the plug-in window from the Plug-In Library view classes.

◆ The Control Manager, which exists in every Process level, handles the communication between DAE and your GUI. Likewise, the Control Manager is usually your entry point to the GUI.

◆ Additionally, you may need to directly access view classes that are present in the GUI window. Usually this is required for display elements, such as custom meters or graphs.

So, in short, the GUI is formed at run-time from the view classes using a description in the Resource Fork. Subsequently, the Control Manager is used to interact with the GUI.

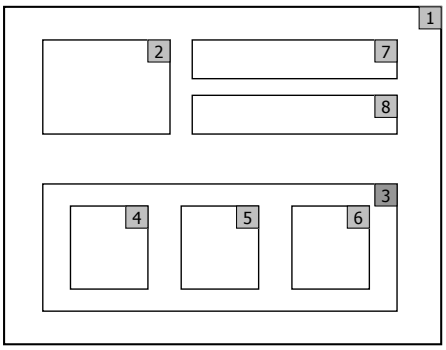
Understanding the View Classes

In the plug-in Library architecture, user interface displays are built out of rectangular graphic elements called *views*. A view is merely an instance of any plug-in view class. Views serve a variety of different purposes. Some will be visible and others will not. There are a variety of different view types. There are views that display text, buttons, sliders, and pop-up menus. Another non-visible view might simply enclose a group of visible views to alter their collective functionality, such as a group of radio buttons.

Each of these view types is implemented as a class in the Plug-In Library, all of which descend from the base class `CPlugInView`. Each view has a rectangle that specifies where it will be located when it is attached to a window. This rectangle is expressed in the window's local coordinate system. In addition

each view has a four-character identifier, specified by an `OSType`. A view is also capable of drawing itself (if it needs to), and responding to user input such as keystrokes and mouse-down events.

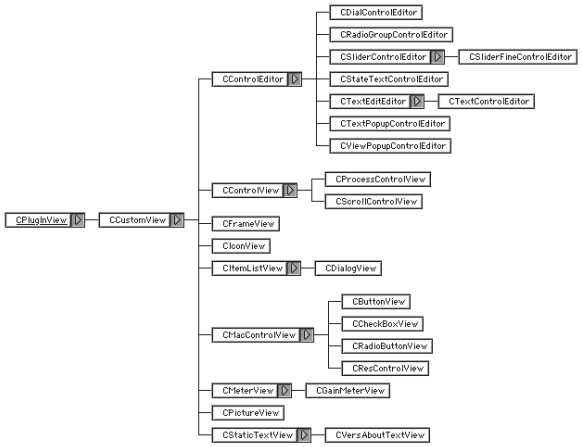
Views can be nested, in a tree-like fashion -- and often are. Any view totally enclosed within the rectangle of another, is said to be a *subview* of the larger *superview*. Internally, view objects contain links to their superview and their *sibling views*; sibling views are views which share the same superview but don't enclose each other. The nesting of views can be arbitrarily deep.



This nesting has been illustrated pictorially. For example, view 2, 3, 7, and 8 are all sibling views. View 3 is the superview of views 4, 5, and 6; thus, they are all the subviews of view 3.

CPlugInView Hierarchy

The plug-in Library provides a handful of standard view elements for you to use in the creation of your user interface. For instance, `CPictButton` provides button functionality using graphical images for its different states. In addition, you have the option to extend these classes and create entirely new GUI elements. To better understand the functionality of these views, let's examine the `CPlugInView` class heirarchy.



You may notice that `CPictButton` is not found on this inheritance tree. This tree only shows the core view classes, and in fact, `CPictButton` inherits its functionality from `CControlEditor`.

Control Editors

A Control Editor is a visible object that is manipulatable by the plug-in user. For example, sliders, knobs, dials, buttons, and radio buttons would all inherit from `CControlEditor`. Control Editor can be linked to other Control Editors, such as a text view that could display and edit the value of the control.

Process Control Views

Process Control Views provide the linking mechanism between multiple Control Editors back to the Process. In previous versions of the SDK, these views had to be explicitly defined by the developer. With more recent versions of the Plug-In Library, the Control Editors generate them automatically as needed. Really, you need to know little about them. Just keep in the back of your mind that they exist.

CDialogView

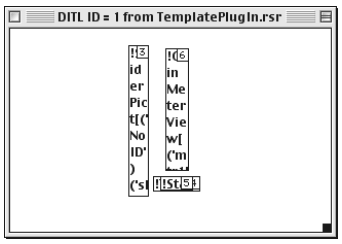
`CDialogView` is also a `CPlugInView`. Yet, it provides special functionality: it forms is the superview of all the subviews for a plug-in window. Embedded functionality allows it to read the Resource Fork information of the plug-in. By parsing the layout data, it's able to locally instantiate all of its subviews. It then initializes them and attaches them appropriately to the view tree it creates below itself. If you are utilizing the Effect Layer, these details can be ignored; you just need to inform the Effect Process which Resource number to reference.

Laying Out the GUI Window

Now, the Resource Fork information that the `CDialogView` parses will be examined in greater detail. Two elements in the Resource Fork are necessary to define the layout and geometry of your plug-in window.

DLOG Resource This describes the dimensions of the plug-in's rectangular window, which initializes the size of the `CDialogView` superview. It also contains a reference to a DITL resource it is matched to. A single plug-in can have multiple DLOG and DITL Resources to define different windows for its separate Process Types.

DITL Resource This resource is the layout map used to create your plug-in window. In reality it's simply a data structure -- a list of items representing the graphic objects which will appear in the plug-in window. Each item has an associated item number, text field, rectangular dimensions, location, and a state (enabled or disabled). When viewing and editing this resource in a program such as ResEdit, it is presented in a graphical nature showing the position and layout of all the elements. This is intended to simplify GUI development by allowing you to manipulate the placement of elements by hand. Unfortunately, as you might determine from this ResEdit screen capture, this is hardly a WYSIWYG system.



Adding DITL Items

Views are specified with a DITL Resource item. The text in the DITL item is interpreted to determine the appropriate view class to instantiate. The size and position of the DITL item are directly translated into the size and position of the view. In addition, each item has an item number. View objects themselves do not use this item number, but the ordering of the DITL items is important during the view-tree instantiation process. The rule for item numbering follows:

- *The item numbers of subviews must be larger than the item numbers of the views that enclose them.*

"Static Text" DITL Items

Typically most views in a DAE plug-in rely on a "Static Text" DITL Item. The static text of the DITL item is then used to define the view class it represents, along with any additional parameters it may need. For those of you with varied UI development backgrounds, this system might appear a bit odd and goofy at first; and, in fact, you would be right. This is part of the "historical background" that was mentioned at the beginning of the chapter. Just understand that this system was adopted to extend the ResEdit-ing process so that custom view classes could added to the window. Really it's quite simple and straightforward as you'll eventually see.

The static text within the DITL item has the general form:

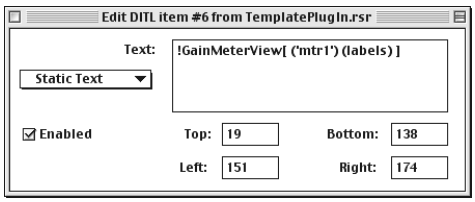
```
!ViewType[(Identifer) ParameterList ]
```

Where the separate pieces are as follows:

```
ViewType =      Picture             // CPictureView
              | StaticText          // CStaticTextView
              | Slider              // CSliderControlEditor
              | TextPopup           // CTextPopupControlEditor
              | RadioGroup          // CRadioGroupControlEditor
              | TextEditor          // CTextControlEditor
              | ItemList            // CItemListView
              | ProcessControl      // CProcessControlView
              | "CustomViewName"    // A Derived CCustomView


Identifier =   "OSType"


ParameterList = (parameter1) (parameter2) (parameter3) (parameter4)...

where,

parameterX =  | OSType             // e.g. 'abcd'
              | character          // e.g. 'a' can include escape seqs.
              | string             // "hola" can include escape sequences
              | number             // (-)1234  Signed long integer
              | constant           //      normal
                                          |clicked
                                          |disabled
                                          |default
                                          |left
                                          |right
                                          |top
                                          |bottom
              | command            // e.g. "getString(resourceId,index)"
                                   // = Include the indexed string from
                                   // 'STR#' resource 'resourceId'.
```

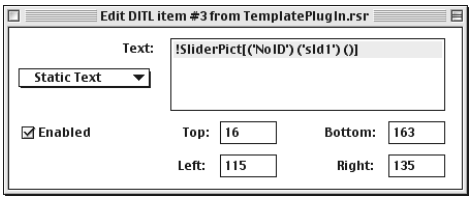
This is probably meaningless for your first time to see it. So, how about a few examples?

`CGainMeterView` creates a LED-style VU meter. To add a DITL item to generate this view, you would need to create the following.



"`!GainMeterView`" denotes that this custom view is of type `CGainMeterView`. The second piece of information is its `OSType` identifier. `'mtr1'` is our arbitrary identifier that can be used to reference the view later within our Process C++ code. The *labels* constant denotes that we wish it to show dB text labels next to the "LEDs." In addition, make sure the item is checked *Enabled*.

Now, how about a slider Control Editor? We'll use the `CSliderPictControlEditor` class. Its DITL view type name is "SliderPict." Control Editors, themselves, don't need identifiers; here, a dummy identifier of 'NoID' is used. On the other hand, they *do* need a Process Control identifier, which is the next parameter. Remember, Process Control Views are generated automatically. This 'sld1' ID links it to other views and allows it to connect back to the control manager. It's probably simplier just think to of 'sld1' as being its identifier. The last parameter is unused in this example.

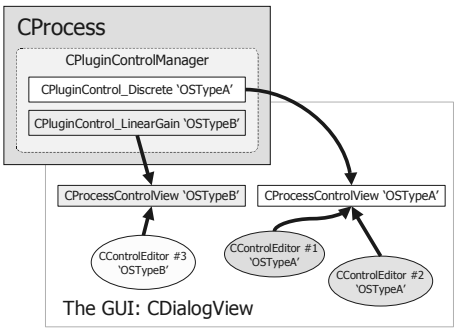


(FYI: This view class also requires that two PICT elements be added to the Resource Fork, with IDs 5000 and 5001. These are included along with the class and are simply pictures of the slider caps.)

Connecting the GUI to the Code

The Control Manager

The Control Mananger, or more precisely the `CPluginControlManager` class, is embedded in the `CProcess` level and manages `CPluginControls` that are added to it. (For those of you previously familiar with the plug-in Library, notice that the nomenclature has changed, and the Control Manager handles "Plug-in Controls" rather than "Process Controls." Hopefully, this naming scheme is less confusing.)



The Control Manager, in a sense, represents the entry point to the controls that the developer can access from his or her C++ code. The Control Editors can be thought of as the visible GUI representation

of these controls, and the Process Control Views as the link between single or multiple Control Editors back to the Control Manager. Process Control Views are actually *non-visible* elements and are generated transparently by the Control Editors.

Adding Your Controls

Now, adding controls to your Process class is almost painless. First, all of your controls must be enumerated starting at '1'. This is important! So, suppose you only had two controls: a dial and a two-state button.

```
enum eMyControls
{
        kControlDial = 1,       // a dial
        kControlEnable          // a two-state button
};
```

Your controls must be enumerated starting at '1'.

Next, ensure that all your Control Editors have been registered at your plug-in's Group level. Assume that you wish to use `CDialControlEditor` and `CPictButton` for your Control Editor classes. `CDialControlEditor` is a registered class by default, so your code might look something like the following. *Warning and disclaimer*: Not all Control Editors have the `RegisterView()` method (see code below). The work of `RegisterView()` might have to be done manually by you. Refer to any Control Editor's `RegisterView()` method for an example of how to do this.

```
void CMyGroup::Initialize(void)
{
        CEffectGroup::Initialize();
        CPictButton::RegisterView();
}
```

Of course, CPictButton.cpp/.h will have to be appropriately added to your project. The DLOG, DITL, and PICT resources must now be editted to contain the dial and button. Five items must be added to the resource: the dimensions of the dialog window via the DLOG, 2 pictures to graphically display a two-state button, and 2 Control Editor DITL items. The static text of the DITL items might look something like:

```
                                !Dial[('NoID') ('dial') ()]

                    !PictButton[('NoID') ('butt') ("BasePictureID" 128)]
```

This also implies that an "up button" picture was added as PICT resource number 128, and a "down button" picture as item number 129.

Adding the PluginControls to the Control Manager is our next concern. To handle the dial, we might want to use `CPluginControl_LinearGain` and for the button, `CPluginControl_OnOff`. The Plug-in Controls use multiple inheritance to "mix-in" different properties to complete a control, usually an "output curve" (log or linear) with a type of units. For example, `CPluginControl_LinearGain` is a mix of `CPluginControl_Linear` and `CPluginControl_Gain`, forming a control which linearly "outputs" values in "dB." As of this writing the usable PluginControls that have been prewritten for you are:

```
CPluginControl_LinearBoostCut    CPluginControl_LogGain
CPluginControl_LinearDistance    CPluginControl_LogRatio
CPluginControl_LinearGain        CPluginControl_LogTime
CPluginControl_LinearLevel       CPluginControl_LogTimeS
CPluginControl_LinearPercent     CPluginControl_List
CPluginControl_LinearTime        CPluginControl_OnOff
CPluginControl_LogFrequency      CPluginControl_Hexadecimal
```

Take a look a the `/PluginLibrary/Controls/` sub-folder of the SDK to get a better idea how this system works. For now, these details should be transparent to us. Now, suppose we'd like the dial to be a gain control with a range of -infinity to 0 dB, and a default of 0dB. We'd also like the button to default to its "on" state. Within the constructor of our Process class, we'd need to add the following code fragment.

```
CMyProcess::CMyProcess(void)
{
        ...

        //kControlDial
        AddControl(new CPluginControl_LinearGain(
                        'dial',
                        "Master Gain\0MastGain\0MstGain\0MGain\0MGan\0MGn\0\0",
                        0.0, 1.0, 0.01, 1.0, true);

        //kControlEnable
        AddControl(new CPluginControl_OnOff(
                        'butt', "Enable\0Enble\0Enbl\0Enb\0\0", true, true);
}
```

It is of great importance to use `AddControl()` on the controls in the same order they were enumerated.

Why? Later on, when we reference the control again, we will need to use this number which is implicitly set by the order of the `AddControl()` calls.

It is also of great importance to pass in unique long names (first name in the string) for each control.

Why? Each time a control is created, it is registered in the Token System (SDS) using the long version of its name. Duplicate long names will cause the system to send the same messages to both controls.

The last *true* arguments of the CPluginControls specify that the controls are automatable. The remaining arguments of the LinearGain control are `minValue`, `maxValue`, `stepSize`, and `defaultValue`. The `stepSize` is important for defining the number of steps that the "continuous" control has; this information is mapped to control surfaces to define the behavior of the control in Fine Control Mode. Fine Control Mode refers to changing the control when the Command key is held down on the control surface. For logarithmic controls, e.g. `CPluginControl_LogFrequency`, a `numSteps` is specified instead of a `stepSize`. Also, for control surfaces, like ProControl, it is ideal to have potential string lengths of 3, 4, 5, 6, 7, 8, and 31 chararcters to describe your control. For both, we have specified several strings that should be sufficient for this range of cases.

At this point:

- The UI should correctly instantiate.
- The on-screen dial and button should be manipulatable and start in their default states.
- They should be capable of recording and playing back automation.
- They should report back to Pro Tools and control surfaces intelligent strings defining their values, e.g. "`On`" or "`-5.2 dB`"

Using the Controls

Of course, now we'd like to have the dial and button manipulate parameters of our Process. In addition, we'd like to be able to save and restore these parameters.

When any control is changed, `CProcess::UpdateControlValueInAlgorithm()` is called. Here, we are given the oportunity to probe the value of the control and make any changes to our state or algorithm. This call might be implemented like the following:

```
void CMyProcess::UpdateControlValueInAlgorithm(long controlIndex)
{
     double gainIndB;
     long enable;

     switch(controlIndex) {
          case kControlDial:
               gainIndB = dynamic_cast<CPluginControl_Continuous*>(GetControl(controlIndex))->GetContinuous();
               ourAlgorithm->SetGain(gainIndB);
               break;
          case kControlEnable:
               enable = dynamic_cast<CPluginControl_Discrete*>(GetControl(controlIndex))->GetDiscrete();
               ourAlgorithm->SetEnable(enable);
               break;
     }
     ...
}
```

Of course, you have the freedom to relay control values to your algorithm however you please. The desire here is only to demonstrate how to examine the value contained within the control. All PluginControls derive from either `CPluginControl_Continuous` or `CPluginControl_Discrete`. It is at this "base" level where you extract the value of the control, as shown above.

It is important to understand that DAE/Pro Tools uses a signed 32-bit integer representation to represent a control (`$80000000` to `$7FFFFFFF`). However, when using the Control Manager this detail can be mostly ignored, and you can think of the value of your control as being in the native format it handles, such as Hz, represented using a *double.* Or, in the case of a discrete control, such as a list, as a discrete set of 32-bit integers, i.e. 1, 2, 3, etc.

To make the conversion back and forth between DAE control values and your native values, a couple more utility functions exist in the PluginControl classes. Theses are:

```
CPluginControl_Continuous::ConvertContinuousToControl()
CPluginControl_Continuous::ConvertControlToContinuous()
CPluginControl_Discrete::ConvertDiscreteToControl()
CPluginControl_Discrete::ConvertControlToDiscrete()
```

Setting Control Values

To set the value of a control use the Process level `SetControlValue()` method. The method, however, only accepts the signed 32-bit integer representation. This is where the conversion ultility functions become useful. For example, suppose you needed to manually update a gain control in your plug-in in response to a user keystroke. The following code fragment acquires the control's current gain value, halves it, and then updates the new control value.

```
     CPluginControl_Continuous *control = dynamic_cast<CPluginControl_Continuous*>(GetControl(kGain));
     double gain = control->GetContinuous();
     gain *= 0.5;
     SInt32 controlValue = control->ConvertContinuousToControl(gain);
     SetControlValue(kGain, controlValue);
```

Hooking Into Meters & Whatnot

What about working with views that aren't user controls? Such as a meter. These we can attach to our C++ code within the `CProcess::SetViewPort()` method. As seen here, we are searching the plug-in's window for a `CGainMeterView` with the identifier of 'mtr1' which has been layed out in the DITL Resource. The method `CPlugInView::FindSubView()` is being utilized. `GetPlugInView()` is an Effect Layer method.

```
void CMyProcess::SetViewPort(GrafPtr aPort)
{
```

```
    // Always start with inherited, in order to attach control to view and init view.
    CEffectProcess::SetViewPort(aPort);

    if (GetPlugInView() != NULL) {
        mGainMeterView
            = static_cast<CGainMeterView *>(GetPlugInView()->FindSubView('mtr1'));
    }
    ...
}
```

Later in the Process, the meter can be updated by directly calling its `SetValue()` method.

```
mGainMeterView->SetValue(someValue);
```

CPlugInControls And Views Reference

CPlugInControls

The first two parameters of all `CPluginControls` are:

1 `OSType id`
2 `const char *name`

The last parameter of all `CPluginControls` are:

Last `isAutomatable`

Keep in mind that the name string must contain a unique name for the long version of the name for each control in the plug-in. The long name of a control is used as the ID to register the control in the Token (SDS) system. Therefore, duplicate names will cause duplicate tokens to be sent to the same controls.

CPlugInControl_LinearPercent

Minimum, maximum, step size, and default values are all specified as `value/100`; i.e., 100% = 1.0.

Parameter List:

3 `double minValue`
4 `double maxValue`
5 `double stepSize`
6 `double defaultValue`

Example Usage:

```
AddControl(new CPluginControl_LinearPercent('revt', "Reverb Type\nRevType\nRev\n", optionList, 0, true);
```

CPlugInControl_OnOff

Parameter List:

3 `long defaultValue` (0 or 1)

Example Usage:

```
AddControl(new CPluginControl_OnOff(
     'swtc', "Low Frequency Band Enable\nLF Band Enable\nLF Band\nLF\n", 1, true));
```

Example String Outputs:

"On", "Off", "1", "0"

CPlugInControl_List

Parameter List:

3 `const std::vector<std::string> &stringList`

4 `long defaultValue`

Example Usage:

```
std::vector<std::string> optionList;
optionList.push_back(std::string ("Hall"));
optionList.push_back(std::string ("Plate"));
optionList.push_back(std::string ("Church"));

// Default for "ReverbType" is "Hall" and is automatable.
AddControl(new CPluginControl_List('revt', "Reverb Type\nRevType\nRev\n", optionList, 0, true));
```

Views

CPictButton

A 3-state button (on, off, disabled) using PICTs for each state.

Register using:

```
CCustomView::AddNewViewProc(
    (StringPtr) "\030PictBackgroundTextEditor", CreateCPictBackgroundTextEditor);
```

Example DITL text:

```
!PictBackgroundTextEditor[('NoID') ('myId') ("BackgroundPictID" 132 "TextColor" 0 0 0 "TextSize" 9
"TextFont" "geneva")]
```

CBackgroundPictView

A text entry box, with specifiable text color, text size, and text font. In addition, a PICT image resource can be placed in the background of the field. If the `BackgroundPictID` keyword along with its associated ID number is omitted **and** the view's superview is a `CBackgroundPictView`, it overlay the text on the `CBackgroundPictView`'s image.

Register using:

```
CCustomView::AddNewViewProc(
    (StringPtr) "\030PictBackgroundTextEditor", CreateCPictBackgroundTextEditor);
```

Example DITL text:

```
!PictBackgroundTextEditor[('NoID') ('myId') ("BackgroundPictID" 132 "TextColor" 0 0 0 "TextSize" 9
"TextFont" "geneva")]
```

CPictBackgroundTextEditor

A text entry box, with specifiable text color, text size, and text font. In addition, a PICT image resource can be placed in the background of the field. If the `BackgroundPictID` keyword along with its associated ID number is omitted **and** the view's superview is a `CBackgroundPictView`, it overlay the text on the `CBackgroundPictView`'s image.

Register using:

```
CCustomView::AddNewViewProc(
    (StringPtr) "\030PictBackgroundTextEditor", CreateCPictBackgroundTextEditor);
```

Example DITL text:

```
!PictBackgroundTextEditor[('NoID') ('myId') ("BackgroundPictID" 132 "TextColor" 0 0 0 "TextSize" 9
"TextFont" "geneva")]
```

CCustomTextPopupEditor

This class implements a typical popup list. By connecting it to one of the various CPluginControls, e.g. CPlugInControl_Discrete or most likely a CPlugInControl_List, the popup control can be completed.

Register using:

```
CCustomView::AddNewViewProc((StringPtr) "\017CustomTextPopup", CreateCTextCustomPopupEditor);
```

Example DITL text (note: the 'revt' ID will connect it to the example CPluginControl_List as presented above.):

```
!CustomTextPopup[ ('NoID') ('revt') ("FontType" 3 "FontSize" 9 "FontFace" 0 "Shadow" 0)]
```

CFrameView

This view class simply draws a black rectange, or partial rectangle.

Register using:

```
CCustomView::AddNewViewProc((StringPtr) "\005Frame", CreateCFrameView);
```

Example DITL text:

```
!Frame[('NoID') (top left bottom right)]
```

The keywords identify on which sides the frame should be drawn. Here, a complete rectangle is formed.

Additional Stuff

Customizing the Cursor

Implement the Process level `DoSetCursor()` call to update the cursor.

Control Color Highlighting

Visual highlight information is used by Pro Tools to indicate that a control is under remote control or is being automated. All controls should support the following color highlights: red, green, blue, and yellow. Most of the plug-in Library Control Editors have built-in support for color hightlighting. Alternatively, a `CHighlightControlEditor` can be placed around (or on top of) another Control Editor, which share a common `OSType` identifier. The Highlight Control Editor will then display a highlighted rectangle around the object when highlighting is enabled.

Pro Tools will request these colors from the plug-in contols based on the following scenarios:

Red:	Write automated
Green:	Read automated
Blue:	Accessible on control surface (stays blue if control is also read automated)
Yellow:	Accessible on control surface and write automated

Digging Deeper Into the Plug-In Library GUI System

If you have absorbed the previous sections, you are now probably competent to build a basic working user interface for your plug-in. The remaining sections detail additional information about the GUI system for those who might need to know a bit more or want to create some custom view classes.

Life of a Plug-In Dialog View

After a Process is instantiated, its `Initialize()` method is invoked. `CProcess::Initialize()` invokes the `CreateCPlugInView()` method which creates a new `CDialogView` and returns its pointer to `CProcess::Initialize()`. `CProcess::Initialize()` then invokes the `InitObject()` method for the view. The inherited method `CPlugInView::InitObject()` eventually gets called, which calls `CreateSubViews()` and `FindSubViews()` for the new view.

Since the view is a `CDialogView`, `CDialogView::CreateSubViews()` is called. This method reads in DLOG Resource from the plug-in's Resource Fork. The Dialog View extracts its rectangle and the ID of the appropriate DITL from the DLOG Resource.

It then calls `CItemListView::CreateSubViews()`, which loads in the DITL resource and creates the subviews specified by its item descriptions. If any of these item descriptions specify CCustomView objects, they are created and initialized with the given parameters.

`CProcess::Initialize()` stores the pointer to the `CDialogView` in the process's `fPlugInView` member, which is referenced in the future whenever the process needs to access its views.

At this point, the Dialog View and all of its subviews are in the detached state - they have not yet been informed of which window they should draw into. Eventually, when a user desires to edit the process's parameters, Pro Tools will create a window and inform DAE about it. DAE will then invoke `CProcess::SetViewPort()` through the dispatch mechanism.

The inherited `CProcess::SetViewPort()` tells the Dialog View (`fPlugInView`) to set its view port. `CPlugInView::SetViewPort()` recursively sets the view port of all of its subviews.

`CProcess::SetViewPort()` then finds all `CProcessControlView`'s and attaches those views back to the `CProcess` via their `SetProcessControl()` method.

Then Pro Tools will issue draw requests and mouse-down events whenever appropriate for the process's edit window.

When the user closes the edit window, Pro Tools will tell DAE that the plug-in's window is about to be closed. DAE will invoke `CProcess::SetViewPort()` again -- this time specifying a `NULL` port. At this point, the plug-in's views will once again be in the detached state.

Creating Custom Views and Controls

RegisterView Method

Keywords

Some custom view classes may use “keywords” in their parameter list to enable or disable features of the view. Keywords are registered in a similar fashion to custom view classes, using `CCustomView::AddKeyword()`. The registered keyword can then optionally be added to an item’s text field in the DITL item to affect the view.

Highlighting Controls

`CProcess::DoTokenIdle()` is responsible for processing highlight tokens (along with all other tokens!) that are sent by DAE to the plug-in. A control number associated with the token is used to determine a `CControlView` object to send the highlight information to. The `CControlView` class stores this highlight information that indicates: if the control is currently highlighted, a highlight color ID (red, blue, green, or yellow), a highlight number, and a text string. The highlight number indicates the ordering among a group of highlighted controls. Your plug-in will typically have a `CProcessControlView` object derived from `CControlView` that receives the highlight information through two functions: `CControlView::SetHighlight()` and `CControlView::SetHighlightInfo()`.

In these two functions, the highlight information is compared with the current information and if it has changed, it stores the new information and calls the new `CControlView::RedrawEditors()`, which tells all the attached Control Editors to redraw.

When an editor, or a subview of the editor, draws itself, it must use the functions `CPlugInView::IsHighlighted()` and `CPlugInView::GetHighlightInfo()` to get information about its highlight status. `CPlugInView::GetHighlightInfo()` allows the view to examine a `short colorID` to determine the highlight color. The `colorID` is one of four constants: `eDAEHighlight_Red`, `eDAEHighlight_Blue`, `eDAEHighlight_Green` and `eDAEHighlightColor_Yellow`. It is up to the view to use an appropriate color.

Suggested RGB color values are:

Red	`{ 0xFFFF, 0x0000, 0x0000 }`
Blue	`{ 0x0000, 0x0000, 0xAAAA }`
Green	`{ 0x0000, 0xDDDD, 0x0000 }`
Yellow	`{ 0xFFFF, 0xCCCC, 0x0000 }`

Refer to the `CHighlightControlEditor` source code for a demonstration on how color highlighting could be implemented in a custom Control Editor class.

SDS Token Automation System

The plug-in Library and the Control Manager make use of the Shared Data Services or the "Token System" to transparently handle the automation of your controls. This section is provided for additional background on how the system operates.

First off, let's briefly discuss SDS. It's a high speed, interrupt safe, asynchronous messaging system that enables inter-component communication within Pro Tools. Plug-ins use this system for automation by 1) sending tokens to write automation data and 2) allowing DAE to handle tokens and update the plug-in while reading automation data.

An automation-enabled control sends a series of tokens to DAE: a touch token, a set token, and a release token. These tokens announce that writing of automation data should commence (if in touch mode), the value that the control should be set to, and notify DAE when the user has finished modifying the control. On playback, DAE also listens for any existing automation data, and updates the plug-ins controls by calling back into the plug-in via the Dispatcher.

Let's take a look at the convoluted sequence of calls that occurs during a typical control update from mouse input. It's important to remember that every Control Editor has an associated Process Control View.

1 The user clicks within the plug-in window. This alerts the DAE app to dispatch the call `PI_DoMouseCommand`.

2 The call eventually propagates to `CProcess::DoMouseCommand()`. The Process level then passes the call to its `CDialogView`.

3 `CPlugInView::DoMouseCommand()` finds the smallest enclosing view of the click, and calls this view's `MouseCommand()` method.

4 If this is a `CControlEditor`, its `MouseCommand()` method first calls `CProcess::TouchControl()` via its `CProcessControlView`. `CProcess::TouchControl()` causes a Token to be emitted.

5 The Control Editor's `TrackClick()` method is invoked via the inherited `CPlugInView::MouseCommand()`.

6 The `TrackClick()` spins in a loop, while the mouse button is depressed, running the following:

> **A** The mouse position is examined, and the ControlEditor's `SetValue()` is invoked if the new control value if necessary.

B `SetValue()` calls the ProcessControlView's `SetValue()` method. This, in turn, calls `CProcess:SetControlValue()` which emits a Token.

C The ProcessControlView's `DoIdle()` is invoked, which calls `CProcess::DoTokenIdle()`.

D `CProcess::DoTokenIdle()` listens for Tokens and updates control graphics accordingly.

E `CProcess::DoTokenIdle()` gives idle time to DAE by calling `FicPersonalityDoIdle()`.

F DAE will handle updating controls if needed by calling `CProcess::UpdateControlValue()` through the Dispatcher; this ultimately updates the Control Manager and makes the call to `UpdateControlValueInAlgorithm()`.

7 `CProcess::ReleaseControl()` is called via the Control Editor's Process Control View; this emits a "Release" token.

It is interesting to note that the Control Editors never update themselves directly. All updates result from DAE bouncing back a Token, where it is handled in the `CProcess::DoTokenIdle()` method. Also, keep in mind that the token system is asynchronous, so you cannot rely on the order and time of tokens arriving.

Chapter 6: Control Surfaces And Page Tables

Overview

Control Surfaces

A tactile, external hardware control surface (CS) can be used to control a plug-in's control parameters. A control surface is often preferred over the mouse, or in conjunction with the mouse, as a means of adjusting parameters due to its accessibility and tactile feel.

There are a few general purpose control surfaces on the market, and a few advanced CS's such as Mackie's HUI, and Digidesign's own D-Control, ProControl, Control|24 and Digi002. Currently, all supported CS units use MIDI to communicate with the host CPU, whether they might be connected via MIDI cables, Ethernet, or FireWire (IEEE-1394). Each of the currently supported surfaces has been given its own "Controller Personality File," which resides in the `DAE:Controllers` folder. When a user enables a given MIDI Controller Personality file in the Pro Tools Peripherals menu, the controller can access and edit plug-ins via defined pages for parameter editing.

Pro Tools will support up to four control surfaces at one time, but only the first enabled peripheral can be used for plug-in editing. Controller types can be mix-and-matched (e.g., you could have a DC16 and a CS-10 work in tandem, but only the first enabled unit can be used for plug-in editing).

Page Tables

Abstractly, a page table is a static mapping of a plug-in's controls to the interface of the control surface. Since a plug-in may have many more controls than the control surface can accommodate at a given time, the controls may be split across several "pages" that the user can freely switch between. More concretely, a set of page tables is simply a set of single dimensional arrays. Each slot of the array corresponds to a particular rotary encoder or push-button of the control surface. By inserting control indices into the elements of the array, a plug-in's controls are mapped to particular tangible controls of the CS. Page tables are stored as XML data created by the PeTE (Page Table Editor) application available on the developer.digidesign website. In the following sections, page tables may be referred to as XML or legacy. XML will refer to the current system with page tables created by PeTE. Legacy will refer to the old method of creating page tables using .r files and compiling them into the Mac resources of the plug-in. Legacy page tables are now deprecated as of the first Intel-based Mac releases of Pro Tools, but they will continue to work in earlier versions of Pro Tools. As XML page table support was introduced in Pro Tools 6.4, plug-ins that support Pro Tools versions prior to 6.4 will need to define both page table formats.

The following sections describe the various interfaces that the supported control surfaces provide for modifying plug-in parameters. Later, the specifics of implementation of page tables is described.

Supported MIDI Controllers

JL Cooper CS-10, CS-10/x

The basic CS-10 unit contains eight 100 mm faders and associated channel switches, Transport switches, several other switches, a Jog/Shuttle wheel, and 6 pots. The CS-10/x expander unit has faders and switches only. Pro Tools supports up to 3 expanders, for a total of 32 faders for this device type. The CS-10 uses the 6 pots for plug-in editing; the faders are used for Pro Tools volume faders only. A 2-digit LED display tells the user which plug-in page the user is editing ("P1," "P2," etc.).

Penny & Giles DC16/MM16

These units incorporate 16 continuous belt faders with imbedded LED indicator controls, and associated channels witches, as well as several other global status switches and a jog/shuttle wheel. The use of fader belts allows a "constant update" mode for the user similar to motorized faders, since they are continuous encoder belts, and they are always at the fader "null point." The DC16/MM16 also incorporate several dedicated function switches, transport switches, and a jog/shuttle wheel.

In addition, a small LCD gives the user information about the current functions of the belts and switches, and is the only one of the three controllers supported today that will tell the user which plug-in control they are currently editing by name. The name of the control appears in the LCD when the switch above the belt in question is touched, or when one of the belt faders is moved. The DC16/MM16 uses the 16 belt faders for pi editing.

Peavey PC 1600

This extremely affordable unit offers sixteen 60 mm faders and associated channel switches. It has a small LCD and a data wheel. With the present firmware for the unit, this display is committed to showing the present "patch" held in the PC 1600's memory, and the data wheel cannot be presently accessed as a separate control to be used for job/shuttle. The patches for the PC 1600 unit are downloaded to it automatically by Pro Tools when the unit is enabled as a CS peripheral. The PC 1600 uses the 16 faders for plug-in editing.

CM Automation MotorMix

MotorMix is a Controller/Motorized Fader control surface for Digital Audio Workstations, manufactured by CM Automation. Motor Mix features 8 rotary pots, a rotary encoder, 68 switches and a 40 by 2 LCD. MotorMix is a control surface to provide tactile control for entering fader moves and mutes into Pro Tools to control levels, mutes and plug-in parameters in real-time. By using the bank select switches and the LCD, Motor Mix can control systems with an unlimited number of channels and many plug-ins in each channel. Pro Tools uses the Mackie HUI page tables for the MotorMix. Therefore, in the following sections, that which applies to the Mackie HUI will apply to the MotorMix as well, including diagrams of the HUI display. For editing plug-ins, the MotorMix has four horizontally aligned switches, numbered left to right, 1 through 4, above four horizontally aligned encoders, numbered left to right, 5 through 8. Looking at a MotorMix, you'll notice that there are 8 physical switches and 8 physical encoders. However, only the even switches and encoders affect the controls of a plug-in.

Mackie Designs HUI

This advanced unit has been designed specifically for Pro Tools 4.x and also provides a powerful control surface interface. Many functions on Pro Tools can be accessed with the click of a single button on the

Mackie HUI. It includes eight 100mm professional-grade motorized faders and a single 4-character LED "scribble strip" on each channel for updating the Channel ID on the fly and for displaying channel attributes. The plug-in editing section consists of four horizontally aligned switches, numbered left to right, 1 through 4, and four horizontally aligned encoders, numbered left to right, 5 through 8; and a vacuum fluorescent alphanumeric display for viewing plug-in parameters.

Ethernet / Firewire / USB Controllers

Digidesign ProControl

ProControl is a modular hardware control surface that adds high-quality tactile mixing and editing capability to Pro systems, as well as a track pad for direct control for menu selection and a switch matrix that also provides alpha entry.

The intent is to provide a high level of integration where the user does not have to context switch off the surface to perform tasks. The system is comprised of a single unit with three sections: Main/editing Section, Fader Section with hi-quality, touch-sensitive motorized faders, and Meter Section with stereo meters for each channel strip. Additional FaderPak expansion units will allow additional fader expansion, in sections of 8 channels each. Up to 3 expansion units will be supported, for up to 32 faders. The alphanumeric scribble strip and plug-in editing displays are hi-quality LED displays (8 character), similar to those found on high-end digital consoles.

The plug-in editing section consists of eight vertically aligned data encoders, numbered top to bottom, 1 through 8, and eight vertically aligned switches, numbered top to bottom, 9 through 16; and an alphanumeric display for viewing plug-in parameters (See picture at right). This gives you a total of 16 possible controls per page. However, if the plug-in only has encoders, it will effectively have only 8 controls per page. Access to pages is governed by 32 access switches, thus ProControl supports up to 32 pages.

Digidesign/Focusrite Control|24

Control|24 is high quality hardware control surface allowing great flexibility and power to Pro Tools systems. The alphanumeric scribble strip and plug-in editing displays allow 4 characters each for the plug-in name and a control's name. Three characters are allowed for the control's value.

The plug-in editing section consists of 24 horizontally aligned data encoders and 24 switches lined up under the encoders. (Picture above shows only 12 so that more detail can be displayed.) However, Control|24's switches and encoders work differently than ProControl: the Control|24 encoders and switches are set up as pairs. In any given control pair, either the encoder OR the switch is used, depending on how the control has been defined in the plug-in. A control defined as a discrete control with two possible values is automatically assigned by Control|24 to the switch of an encoder/switch pair. A control with three or more possible values is automatically assigned to the encoder. If the control is set to the encoder in a pair, the switch in that pair will be disabled. Therefore, whether the plug-in has all switches, all encoders, or a mix, there are 24 controls available per page on the Control|24.

Digidesign Digi002

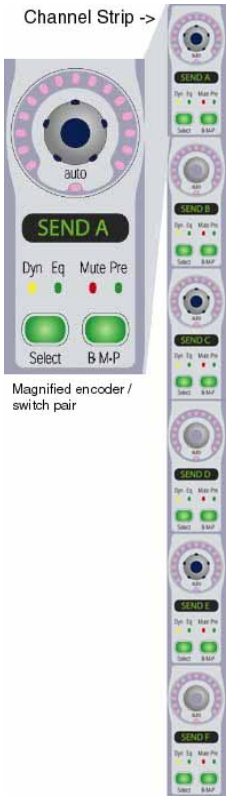
The Digi002 can work either as a stand-alone mixer or as a control surface for Pro Tools LE. Its architecture is based on the Control|24, so many of the features are the same. The alphanumeric scribble strip and plug-in editing displays allow 4 characters each for the plug-in name, a control's name, and the control's value.

The plug-in editing section consists of 8 horizonatlly aligned data encoders and 8 switches lined up under the encoders. Just like the Control|24, the encoders and switches are set up as pairs, where a control with two values is assigned to the switch and a control with three or more values is assigned to the encoder.

Digidesign D-Control

D-Control is a modular hardware control surface that adds high-quality tactile mixing and editing capability to Pro Tools systems. D-Control is a high-end mixing system that includes a center section and one or more fader packs. A fader pack consists of 16 channel strips, displaying track and plug-in information. Further general information about D-Control and how it works is available from the "BuckleyBriefing.pdf" on the developer website.

D-Control has the potential of displaying plug-in controls in four different sections: Channel Strip, Custom Fader, Center Section EQ, and Center Section Dynamics. The Dynamics section actually makes use of two different page table types, while the other three each have one page table type. In all sections, scribble strips are 6-characters wide and do *not* support the Digidesign Extended character set (i.e. special characters).



■ Channel Strip

Rather than having a dedicated plug-in editing section like ProControl or Control|24, plug-in controls are displayed in the channel strip. Each channel strip consists of 6 vertically aligned encoder/switch pairs. This gives a total of 12 controls per page, where the encoders (from top to bottom) are numbers 1-6, and the switches (from top to bottom) are numberes 7-12. Like ProControl, the lower numbered encoder is paired with the lower numbered switch. In the diagram to the right, the button labeled "B-M-P" will control the switch.

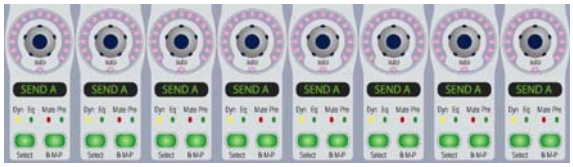
There are a couple of special considerations when making D-Control Channel Strip page tables. First, keep in mind that the strips are at the top half of the control surface. Therefore, it will be more convenient for the user if you place the most frequently used controls at the bottom rather than at the top. Second, when a control is present in the switch of an encoder / switch pair, the "Pre" light will be lit as an indication to the user of the switch's presence. However, the name of the control placed on the switch will not appear in the scribble strip, unless the user presses the Sw Info button. The value of the switch will appear when the switch is pressed, but the name does not appear. Therefore, it is not recommended that you place a switch next to an empty encoder. It is recommended that you either place a switch next to an encoder that makes sense (like a Gain encoder next to a Phase switch) or place the switch control in both the switch and encoder so

that the user will see the name of the control. These are merely guidelines, and not hard and fast rules.

■ Custom Fader

D-Control can be placed in a special mode called Custom Fader to allow more controls of a plug-in to be displayed at once. D-Control takes the currently selected plug-in and displays its controls across 8 of the 16 channel strips such that the plug-in's first control is in the lower left control, its eighth is in the lower right. If there is more than one page of controls, D-Control displays additional pages, up to six, in the the rows above the first. If the user has more than one fader pack, the Custom Faders can spread to more sets of faders.

However, all you need to do as a plug-in developer is create a page table with 16 controls (8 encoder/switch pairs) per page. The diagram below shows the layout of one page. See the diagram to



the right to see how the layout of several Custom Fader page table pages works. Since the one page layout is similar to the ProControl layout (16 controls per page with 8 encoders and 8 switches), D-Control will use your plug-in's ProControl page tables as a default. If you are not happy with how your plug-ins controls appear in this manner, you can override this by creating a Custom Fader page table.

■ Center Section EQ and Dynamics Sections

New to any Digidesign control surface is the idea of a center section. D-Control's center section contains two dedicated areas for EQ and Dynamics plug-ins. Only plug-ins falling into these categories (EQ, Compressors, Limiters, Expanders, and Gates) should implement page tables for these sections. If your plug-in does not fall into these categories, you can skip these page table types. Since the Center Section page table types will also be used in future Digidesign control surfaces, the remainder of the descriptions of these types can be found below in the Center Section Page Tables section.

Digidesign Command|8

The Command|8 is a small control surface, similar to the Digi002, that will work with TDM and LE systems and connects to the computer via USB. The alphanumeric displays are like D-Control in that they are 6 characters wide and do *not* support the Digidesign Extended character set. Therefore, if you have already added 6 character versions of your control names / values, your plug-in will display these properly on the Command|8 as well.

The plug-in editing section consists of 8 horizonatlly aligned data encoders and 8 switches lined up under the encoders. Just like the Digi002, the encoders and switches are set up as pairs, where a control with two values is assigned to the switch and a control with three or more values is assigned to the encoder. The Command|8 uses the Digi002 page tables so if your plug-in already includes Digi002 page tables, it will automatically work with Comand|8.

Digidesign Center Section Page Tables

These page tables are currently used by the D-Control (ICON) and D-Show (VENUE) consoles. The examples in this section will mostly refer to D-Control. Venue developers, please see the D-Show specific documentation for further details.

There is also a new plug-in library API, GetControlValueInfo( ), that works hand in hand with these page table types. You may optionally implement this method for enhanced support of several buttons and status LED's on these sections. We will describe the new page tables first and then move on to discuss GetControlValueInfo( ).

There are three Center Section page table types, defined in FicHWController.h:

```
const OSType cDigiEQPageTable         = 'DgEQ';
const OSType cDigiCompLimPageTable    = 'DgCP';
const OSType cDigiExpGatePageTable    = 'DgGT';
```

The first type is for EQ plug-ins (and supports the D-Control EQ section). The second type is for compressor and limiter plug-ins (and supports the D-Control Dynamics section). The third type is for expander and gate plug-ins (and supports the D-Control Dynamics section). Dynamics plug-ins that include both types of processing should support both DgCP and DgGT page tables.

While future control surfaces will use these same page tables, the physical layout of the controls on these other surfaces may be different than on D-Control. However, our goal is that your plug-ins will be able to automatically work on these new surfaces without any additional work.

It is important to note that these three page table types are different from all other page table types in a fundamental way:

*Each slot in the page table is **pre-defined** for a specific type of control.*
Therefore, your plug-in must *conform to this pre-defined layout.*

The purpose of the Center Section is to give the user a standard interface for EQ and Dynamics plug-ins – no matter what particular plug-in they are using. It allows the user to quickly access the most common controls in their favorite EQ or Dynamics plug-ins. This is different from all other page table types because the only restriction on other page table types is that – for types that have dedicated discrete controls – you cannot place continuous controls in a dedicated switch. However, the user will also know that if there are controls they would like to access that are missing from the Center Section, they can access them through one of the other layouts available on the control surface.

You'll notice looking at the pictures of these sections in D-Control (below) that the only scribble strips are in the center of the section. Unlike the Channel Strip section, there is not a scribble strip to label each control. The control's purpose is physically printed on the control surface. That is why it is imperative that your plug-in conform to the pre-defined layout. You can find the definition of these page table types in FicHWController.h and in the PeTE editor. If the meaning is unclear please contact Developer Services.

Because of the strict definitions of the layouts, it may mean that 1) not all of the controls for your plug-in can fit in these sections, and that 2) there may be controls your plug-in does not have and therefore are blank in this view. For example, let's say your plug-in is a 10-band EQ that does not have individual Q controls on any of the bands. Such a plug-in will be forced to leave off some of its bands, even though all Q controls specific to the bands on the page table are empty. That is fine as there will be another way for the user to display the plug-in on the control surface that will include all of the controls. For example, on D-Control, a user can also view an EQ or Dynamics plug-in in both the Channel Strip and Custom Fader modes which will display all controls. The important point is this:

Do not place a control in a Center Section page tables that does not fit its prescribed definition.

If you do, the control's function will be mislabeled and will cause confusion for the user. The main purpose of the Center Section is to give the user a standard experience when using EQ and Dynamics plug-ins.

Continuing that train of thought, you should only implement one page for these Center Section layouts. This is different from the other page table types, where it is expected to implement as many pages as necessary to give access to all controls in the plug-in. In the case of the Center Section layouts, you

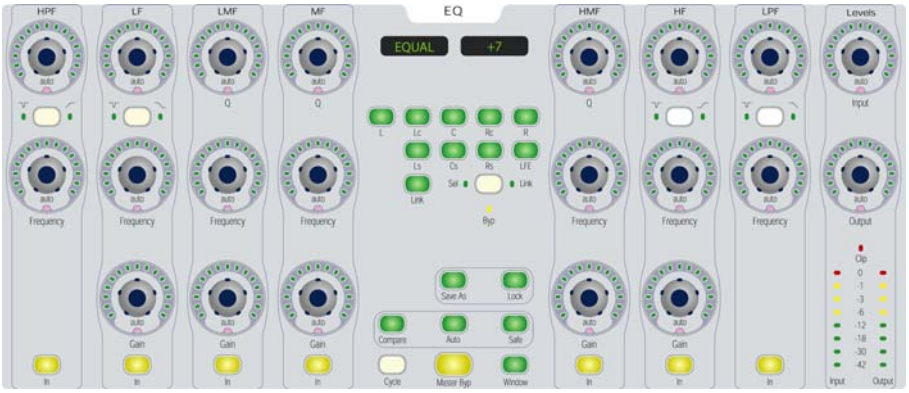
should only define one page, except in the case when the plug-in has separate controls for each channel (Left, Right, Center, etc.).

More than one page in Center Section layouts is allowed **only** *if the plug-in has separate controls for each channel.*

For example, if your EQ plug-in allows the user to change the EQ differently for the left and right channels, then you would implement two pages for the DgEQ page table. You'll notice in the pictures below for the EQ and Dynamics sections of D-Control, there are buttons labeled for channel selections (L, LC, C, RC, R, etc.). The control surface will automatically map the pages to the buttons, according to the standard order for surround channels. For example, in a stereo EQ, Left controls should be in the first page, and Right controls in the second. D-Control will automatically map page 1 to the L button and page 2 to the R button.

We cannot emphasize strongly enough that trying to fill in all of your controls into these layouts, whether it is by creating extra pages, or by filling in empty slots, will only serve to confuse the user. You must adhere strictly to the guidelines given for these sections.

Let's look at D-Control's EQ section more closely.



It supports a maximum of seven bands of EQ, each a vertical column of controls and labeled from left to right as follows:

- HPF (High-Pass Filter, nominally a high-pass filter or low frequency notch filter)
- LF (Low Frequency, nominally a parametric EQ or low frequency shelf filter)
- LMF (Low-Mid Frequency, nominally a parametric EQ)
- MF (Mid Frequency, nominally a parametric EQ)
- HMF (High-Mid Frequency, nominally a parametric EQ)
- HF (High Frequency, nominally a parametric EQ or high frequency shelf filter)
- LPF (Low-Pass Filter, nominally a low-pass filter or high frequency notch filter)

Each of these bands has a Q/Slope control, Frequency control, and an In Circuit/ Out of Circuit button. Five of the bands have an additional Gain control. Four of the bands have an additional EQ type selector switch, each surrounded by a pair of EQ type LED's. Input and Output Level controls are also available, as is a multi-channel Link button in the middle section.

If an EQ plug-in implements a band that does not have an In/Out Circuit control or a Type control, but wants the related LED's to light properly, please see the discussion of the `GetControlValueInfo()` *API in the "Deeper Into the Plug-In Library" Appendix.*

The topmost rotary encoders in the HPF, LF, HF, and LPF bands are not labeled but they are indeed Q / Slope controls. The EQ type selector switches located between the Q/Slope and Frequency knobs control

the type of filter on that band. Thus they define the behavior of *all* controls in that band (and not just the unlabeled Q/Slope control).

Starting with Pro Tools 6.9 and the 6.9 Plug-In SDK, new controls have been added to the EQ page table type. Previously, the page table only defined one slot in each band for Q ***and*** Slope. Since the page tables only allow one control to be defined in any given slot, it was determined that this design was insufficient to allow for plug-ins that may require separate controls for Q and Slope in the same band. Therefore, the page table has been updated with one additional slot per band (called xxx_Q_Or_Slope_Alt in FicHWController.h) to handle this case. In order to use this feature, a plug-in must respond to the `eDigi_PageTable_UseAlternateControl` selector in the `GetControlValueInfo()` method. When the band type is changed, the control surface will call into the plug-in with this selector to determine if the control in the "Alt" position should be used. Please see the discussion of the `GetControlValueInfo()` API in the "Deeper Into the Plug-In Library" Appendix for a detailed example.

If your plug-in supports fewer than seven simultaneous bands of EQ, you have some options for where to place them. We recommend the following placement guidelines so users have consistency with various EQ plug-ins.

- If you only have one band of EQ, please use the LF band.
- If you have one to four fully parametric bands, please use LF, LMF, HMF, and HF (starting from left to right). (Skip the MF band.)
- If you have up to two shelving filters and two parametric bands, please use LF (LF shelf), LMF (para), HMF (para), and HF (HF shelf). (Skip the MF band.)

The DgEQ page table layout is defined by EDigi_EQPageTable in FicHWController.h. If you are not using PeTE to create your page tables, i.e. you are using legacy page tables, follow the format laid out in this file when creating your .r file. Otherwise, follow the labels in PeTE in the EQ layout. Note that legacy page tables should only be used in order to support version of Pro Tools prior to 6.4. Otherwise, XML page tables are required.

Note that this layout includes a few additional controls not in D-Control's EQ section to accommodate future Digidesign control surfaces. These extra controls are:

```
eDigi_EQPageTable_LMF_Type
eDigi_EQPageTable_MF_Type
eDigi_EQPageTable_HMF_Type
```

Now we'll look at D-Control's Dynamics section. Keep in mind that the D-Control's Dynamics section displays page tables for both the Compressor/Limiter page table type and the Expander/Gate type. Therefore, the function of certain rotaries and switches differs depending on which page table type has been loaded. In these cases, there is a LED to indicate the current function of a rotary or switch.

Several rotary encoders have alternate uses while others are always dedicated to one function. The entire set of controls is as follows:

- Knee / Hysteresis
- Ratio / Range
- Attack
- Release
- (Makeup) Gain / Hold
- Threshold
- (Key) Low Filter Frequency
- (Key) Low Filter Q/Slope
- (Key) Low Filter EQ type selector (with notch and high-pass filter LED's)
- (Key) Low Filter In Circuit / Out of Circuit
- (Key) High Filter Frequency
- (Key) High Filter Q/Slope
- (Key) High Filter EQ type selector (with notch and low-pass filter LED's)
- (Key) High Filter In Circuit / Out of Circuit
- Input Level
- Output Level
- External Key (middle section)
- Key Listen (middle section)
- Multi-channel Link (middle section)

The compressor/limiter page table, DgCP, supports all the controls above except Range, Hysteresis, and Hold. (As you create the page table in PeTE, this is clear.)

The expander/gate page table, DgGT, supports all the controls above except Knee and Makeup Gain. It does support both Ratio and Range. The user presses the Page button to select between them.

A plug-in can support both DgCP and DgGT page tables. Again, the user presses the Page button to select between them. If a plug-in supports both of these page tables and the DgGT page table includes support for both Ratio and Range controls, pressing the Page button will switch between all available controls as follows:

DgCP -> DgGT with Ratio -> DgGT with Range (and all other controls unchanged)-> DgCP ->...

The layout for DgCP is defined by EDigi_CompLimPageTable.
The layout for DgGT is defined by EDigi_ExpGatePageTable.

Both may be found in FicHWController.h. Again, if you are using PeTE, most of the details above can be ignored as long as you follow the labels in PeTE for each control in the Comp/Limiter and Expander/Gate layouts.

In addition to these page tables, a new plug-in library API has been added to provide enhanced support for some LED's and buttons in D-Control's EQ and Dynamics sections. `GetControlValueInfo( )` allows the application to query a plug-in for the "meaning" of its control values. See the CProcess section of the Appendix "Deeper Into the Plug-In Library" for more specific information on how to use this API.

Plug-In Page Table Guidelines

This section is intended as a guide in setting up defined 'pages' using some general rules. However, due to the sheer number of variables, it is simply not possible to account for all scenarios. But by following these suggestions, a TDM plug-in developer should find these guidelines useful in setting up their own plug-in pages. Moreover, it is hoped that a consistent and somewhat standard mapping topology will be realized across the broad range of plug-ins and control surfaces.

Here, we are primarily concerned with the number of controls on a control surface (CS) available for plug-in editing; and secondarily with the layout of controls provided for plug-ins. Accordingly, we need a method of mapping a plug-in's control parameters to a CS, and we need to take into account the varying numbers of controls available on a CS. Using 'page tables', from a software point of view, a plug-in's control parameters can be mapped to a CS. Each page of the page table describes which of the plug-in's parameters will be accessible from the CS's controls that are used for plug-in editing. Multiple pages are needed in the case where a CS has fewer controls available than the actual number of controls on a plug-in.

We begin by stating guidelines that should be followed when mapping a plug-in's control parameters to a CS. At the end of this chapter, you will find the technical details of creating page tables for your Plug-in.

The following guidelines are for simple, generic control surfaces. More advanced CSs, such as the Mackie HUI and ProControl, have some guidelines of their own which are listed after these.

General Guidlines

- Map a plug-in's controls from left-top to right-bottom sequentially onto each page.

Follow the layout of the plug-in GUI as closely as possible, allowing the controls to sequentially map to the Control Surface in the order specified above. In so doing, the CS controls will match the plug-in GUI; in the sense that by counting the location of a given control on the PI GUI, one should be able to grasp the corresponding slider or pot on the CS.

Note that Master Bypass, located in the plug-in's floating inserts window, should nearly always be placed as the first control on the first page. The only time this guideline might not be followed is if the plug-in has a particularly favorable layout for the control surface, and where this placement of Master Bypass would disrupt it. Also, on some control surfaces a dedicated bypass is already provided, in which case the Master Bypass should not be mapped into the page table.

- Related control parameters should be grouped together on the same page.

Controls that are often 'adjusted' with other similar or related controls should be mapped to the same page. This enhances the users ability to tweak related parameters and alleviates unnecessary paging.

- Related control parameters should not be split across pages.

This follows directly from #2 above. If some closely related controls cannot all fit on the same page, it is better to leave some blanks (i.e., unused pots, sliders, or switches) and move onto the next page where they can be adjusted together.

As a hypothetical example, let's say a control surface has 5 sliders, and we are mapping an EQ PI with 6 parameters — a low, mid, and high frequency band which has gain for each band. It would be best to map them to the CS as follows on page 1, from left to right on the CS: low freq, low gain, mid freq, mid

gain, blank. Then map the remaining two parameters onto page2: high freq, high gain, blank, blank, blank.

- Equivalent left and right stereo parameters should remain on the same page.

Since adjusting the left or right parameter of a stereo PI has considerable impact on the sound field, it is important that equivalent left and right stereo controls remain on the same page. Contrast this to placing the left parameters on one page, and the right parameters on another which is not desirable. This rule also changes according to the controller's layout. As an example, the CS-10 has 6 pots for PI editing arranged in a matrix of 3 rows x 2 columns. From left to right, the pots in row 1 are numbered 1 and 4. In row 2 the pots are numbered 2 and 5. Finally, the pots in row3 are numbered 3 and 6. A layout for L/R controls should be mapped param1L = control 1, param1R = control 4, and so on.

- Repeat control parameters on pages where it makes sense to do so.

In some situations, it is desirable to have access to the same control on many pages. For example, this might mean having an output and/or input gain control available on each page of an EQ PI — since EQs change the overall gain. This is especially desirable if there would otherwise be blanks (i.e., unused pots, sliders, or switches).

Advanced Control Surfaces

The Mackie HUI, Digidesign's D-Control, ProControl, Control|24, Digi002, Command|8, and similar advanced control surfaces require some modification to the general guidelines to best make use of their interfaces; this is where the benefits of a CS can be fully realized! All of these CS's have alphanumeric displays that can display individual plug-in control names and values. Therefore, a sequential linear mapping (top/left to bottom/right) as outlined above under the generic guidelines no longer has to be followed. When possible, it will be more beneficial and user-friendly to follow standards that are already in place on existing processors and consoles in the industry.

- Follow industry standards when possible.

There are good examples already in place in the audio industry as to what has become reasonably standard on mixing consoles and processing gear. It is best to follow these "standards" when possible. For instance, on mixing consoles with vertical layouts, an EQ is generally laid out such that the lowest frequency band is closest to the user, toward the bottom of the console, and the highest frequency band is furthest. With a rack mount EQ processor, the lowest band is usually on the left side, and the highest band is usually on the right. Continuing with the EQ example: On a mixing console, if there is a gain control associated with each frequency band of the EQ, it is usually placed above the band it controls. On a rack-mount EQ, the gain will usually be placed to the right of the band it controls.

How does this relate to mapping PI parameters? Well, the Mackie HUI, Control|24, Command|8 and Digi002 might be visualized as a 'rack mount processor' abstraction — with its horizontal layout for controlling PI parameters; and D-Control and ProControl might be visualized as a 'mixing console' abstraction — with its vertical layout for controlling PI parameters. Thinking in these terms can help in constructing effective page tables.

- PI switches should be mapped to switches on the CS. Also, switches should be placed as close as possible to the rotary encoder it controls, when applicable.

It is most intuitive to place a switch as close as possible to the parameter it operates on. There are two main groups for handling switches: CSs whose switches are separate controls, like the D-Control, HUI

and ProControl, and CS's whose switches and encoders occupy the same control space, like the Control|24 and Digi 002/Command|8.

For CS's like the HUI and ProControl, this may mean there will be blank (unused) switches next to data encoders that don't have an associated "active" switch parameter. Also note that some switches may not be associated with a parameter directly, but may have some constituent relation — such as the association between a 'gain' control and a 'phase' switch; where the switch may conveniently appear next to or above its associated data encoder.

With the Control|24 and Digi002 control surfaces, a switch and encoder occupy the same controller space. If the control assigned to that space has only two values (e.g., "on" and "off"), then the control is assigned to the switch automatically. If the control has three or more values, either discrete or continuous, then it is assigned to the encoder, and the switch is disabled.

Implementing Page Tables

Page tables can be created and edited using the Page Table Editor (PeTE) application available on the Developer website. PeTE generates an XML file that can then be used in both Windows and Macintosh plug-in projects. On the Macintosh, the XML is compiled into the plug-in as a resource using a PageTables.r file. On Windows, the XML is compiled into the plug-in as a resource using a PageTables.rcx file. See the PeTE documentation as well as the example plug-ins provided in the SDK to see how this is done.

PeTE will automatically import legacy page tables from existing plug-ins

At a minimum, generic page tables of sizes of 5, 6, 8, and 16 should be provided, along with page tables for D-Control (all applicable types), Procontrol, Control|24, Digi002 / Command|8, and Mackie HUI.

Though not required, it is also possible to set up other page sizes. This may be advantageous if your plug-in lends itself well to a particular layout scheme. In this case, if a newer CS appears on the market, your plug-in will be ready to support it. If your plug-in does not specify a page table explicitly for the newer CS, it will still support it; but the mapping will be based on a Generic page table, first of the same size as the new CS, and if that is not found, then of page size of one. Note that the ordering of controls in dialog boxes, e.g. Plug-In Automation dialog, is determined from the one-control-per-page layout. Therefore, for these reasons, it is important that a Generic page table of size one is defined!

There are ten different resource IDs for the ten currently supported page table types. The following table outlines these IDs.

Page Table Type	ID
Generic	PgTL
D-Control Channel Strip	BkCS
D-Control Custom Fader	BkSF
Center Section EQ	DgEQ
Center Section Comp/Lim	DgCP
Center Section Exp/Gate	DgGT
ProControl	PcTL
Control 24	FrTL
Digi002	HgTL
Mackie HUI	MkTL

After compiling a plug-in project, a resource will be created containing the XML data. If after editing page tables in PeTE, you want to quickly test your changes, you can do the following:

- After editing your page tables in PeTE, save the file with a higher version number than the last saved file
- Copy the XML file created by PeTE to the Plug-Ins folder and drop it next to your plug-in
- Change the name of the file to <PluginFilename>.xml. <PluginFilename> must match exactly the plug-ins filename. For example, if your plug-in's file name is EQ dbg.dpm, the XML file must be called EQ dbg.dpm.xml
- Launch Pro Tools. DAE will find the file, and as long as its version number is higher than the XML compiled into the plug-in, it will load the file's page tables.

Although page tables and control surface support do require a lot of attention to detail, plug-ins that are control surface aware are essential to paving the way for the next generation of powerful digital audio tools. And we believe that this has far-reaching benefits for everyone, including plug-in developers!

Verifying Page Table Layouts: The "Hidden Pop-Up Menu"

You can verify the page tables created in your plug-in in Pro Tools with a “hidden” developer debug Page Tables popup menu. To verify the page tables, first include the YourPageTables.r file in your project and compile the plug-in. Then, after launching Pro Tools, instantiate the plug-in, hold down the Command-key (Ctrl-key in Windows) and mouse-click the Automation button in the plug-in window to display the menu.

Category The Category menu item has a submenu listing the names of possible categories. Any category that the plug-in belongs to will have a check mark next to it. In addition, appended to the name of the category is an indication of whether that category can be bypassed, and if so, the control number (#) and control name of the associated bypass control. Also appended is the number of the first page on which a control associated with the category can be found. This page number is based on whatever the current page table type is selected in the "TableType" menu item. For example: Delay (can bypass, control #9, Master Bypass) (first page #1)

Table Type Sets the type of control surface page table.

Page Size Sets the number of controls per page. The identifier "custom" is shown next to page sizes that have been specifically implemented (which at minimum, should appear next to page sizes of 5, 6, 8, and16!). The identifier "default" is shown next to page sizes that do not have specific support in your page table file. Note that the Mackie and ProControl table type will automatically set this to 8 and 16, respectively.

Control Name Length Sets the number of characters to be displayed in the control's name (shown in the Page menu below). The identifier "expected length" appears next to lengths that should be specifically addressed (3, 4, 5, 6, 7, 8, and 31). If you use the XML page table system, the names can be specified in the PeTE application. Otherwise, the function GetControlNameOfLength() is responsible for providing names with these lengths.

Control Value Length Sets the number of characters to be displayed in the control's value (shown in the Page menu below). The identifier "expected length" appears next to lengths that should be specifically addressed (4, 5, 6, 7, 8, and 31; also, 3 is used for ProControl switch states). The plug-in Library call `GetValueString()` is responsible for providing values with these lengths.

Highlighted Page Highlights the selected page in the plug-in window in Pro Tools.

Highlight Color Sets the highlight color. The highlight color can be: red, green, blue or yellow. Note that at minimum, plug-in's should support these four colors of highlighting!

Page X The actual page table layouts are shown here. The following information can be seen in this menu item.

- The control's name, as returned from the XML page tables. If the table type is Mackie, then the length will be 4 (unless overridden by changing the "Control Name Length" menu item). If the table type is ProControl, the length will be 3 (again, unless overridden, but usually there is no point in doing so). Also, with ProControl set as the table type, the special ProControl symbols will appear here if they are part of the name (however, note that they are small and can be difficult to see). Finally, if the table type is default, the name will be shown with the number of characters as specified in the "Control Name Length" menu item.

- The control's value is shown next. Both the Mackie and ProControl table types will automatically set the control's value length to 4 and 3, respectively. Otherwise, this can be set with the "Control Value Length" menu item. `GetValueString()` is responsible for providing this 'value' information.

- The number of control steps is shown next for continuous and discrete controls. For example, a discrete control will appear as:"(discrete: N steps)", where N is the # of steps of the control.

■ If "(NoL)" is displayed, this simply means that it is a new plug-in, and supports either the XML page tables or the `GetControlNameOfLength()` function call. If "(NoL)" does not appear, then the plug-in is older and you should not expect the control names or values to be optimized — since they are created by just truncating the longer values that the older plug-ins return.

■ If "(highlight)" is displayed, this means that the string will be reverse highlighted when displayed on hardware controllers that support it. Not all hardware controllers utilize reverse highlighting.

■ Next, orientation flags for each plug-in control will be displayed. The format is: "(Value: xxx yyy)," where xxx is either "BMin" (`kDAE_BottomMinTopMax`) or "TMin" (`kDAE_TopMinBottomMax`); and yyy is either "LMin" (`kDAE_LeftMinRightMax`) or "RMin" (`kDAE_RightMinLeftMax`). This text identifies the value of the control's orientation flags, as returned by `GetControlOrientation()`.

■ Also shown is the radial LED encoder-display mode (as returned by `GetControlOrientation()`) assigned to each control (this radial LED surrounds each encoder). The possible modes are: "spread" `kDAE_RotarySpreadMode`, "wrap" `kDAE_RotaryWrapMode`, "boost" `kDAE_RotaryBoostCutMode`, or "dot" `kDAE_RotarySingleDotMode`, along with either "RMin" `kDAE_RotaryRightMinLeftMax`, or "LMin" `kDAE_RotaryLeftMinRightMax` — indicating which side the minimum DAE value is being mapped to.

Color Highlighting Scheme

Note that plug-in controls that are currently controllable on any page of a plug-in will be highlighted in blue when they are the active page on a control surface. Therefore, it is very important to implement the highlighting of controls in your plug-in. Plug-in highlighting is also used with automation. In general, four common colors should be implemented:

Red:	Write automated
Green:	Read automated
Blue:	Accessible on control surface (stays blue if control is also read automated)
Yellow:	Accessible on control surface and write automated

You can use the “hidden” popup menu above to test that your color schemes are working properly.

Control Numbering Layouts

Most of the advanced control surfaces have both rotary encoders (knobs) and switches. The knobs and switches are handled differently by different surfaces. Control|24 has a set of 24 rotary encoders and 24 switches for plug-in editing, and Digi002 has 8 encoders and 8 switches, all set up in pairs. However, both of these control surfaces automatically assign a control with only two possible values (e.g., “on” or “off”) to a switch, while a control with three or more possible values, whether it is discrete or continuous, is automatically assigned to the rotary encoder. Therefore, no distinction is made between control numbers for switches and control numbers for encoders.

The following tables show how the control numbers are arranged for control surfaces which have distinctions between their encoders and switches.

D-Control Channel Strip

Encoders	Switches
1	7
2	8
3	9

4	10
5	11
6	12

ProControl / D-Control Custom Fader Mode

Encoders	Switches
1	9
2	10
3	11
4	12
5	13
6	14
7	15
8	16

Mackie HUI

Encoders	Switches
5	1
6	2
7	3
8	4

Page Table Examples

Following are two examples that will be used to discuss the layout of two Digidesign plug-ins, ProX and 4-Band EQ for Mackie HUI and Digidesign ProControl.

HUI Spreadsheet Format

Mapping the ProX plug-in's 11 controls requires two pages on the HUI.

Page No	Control Number	Switches	Units	Control Number	Encoders	Units	VPOT Mode
1	1	Inv	On/Off	5	Inpt	dB	wrap
	2	Blank		6	Mix	%	0
	3	Blank		7	Dly	ms/sec	0
	4	FB/P	Pre/Post	8	FB	±%	0
2	1	LP/P	Pre/Post	5	LPF	OFF/Hz	0
	2	Blank		6	BPM	bpm	0
	3	Blank		7	Note	*See below	0
	4	Blank		8	Grve	%	0

* These values are: Whol, Wh+, 1/2, 1/2+, 1/4, 1/4+, 8th, 8th+, 16th. These are not units -- instead, they are implemented with `GetValueString`).

Shown above in the ProX Table for HUI are two pages that represent all of the switches and controls on the ProX plug-in. Control Numbers 1 through 4 are switches, and 5 through 8 are continuous controllers on the HUI.

In general, the format of the table shown above should be straightforward. On Page 1, switches are shown in columns (i.e., the first four Switch items on the left side of the table: mapped to Control Numbers 1 - 4); and similarly for the continuous controllers (i.e., the first four Encoder items on the right side of the table: mapped to Control Numbers 5-8). Likewise, Page 2 follows the same format. Parameters are given control and switch name abbreviations that are appropriate for HUI; while unused switches or encoders are represented by blanks. Switches show both their "engaged" and "disengaged" state, respectively, separated by a slash. The continuous controller units column may also show abbreviations when appropriate, and will often include standard metric scaling symbols (e.g., m for milli, k for kilo, etc.) — though technically, these are not units, and should be dealt with separately as follows.

The units for any parameter will usually remain the same, except for possible abbreviations if it has to be shortened (to fit in the required length). However, the associated metric scaling symbol may change depending on the scale as a control is varied throughout its range. Therefore, when appropriate, append any metric scaling symbols to the end of values in the function: `GetValueString()`, rather than including them in the `GetControlUnitsString()` function.

ProControl Format

Mapping the ProX plug-in's 11 controls requires one page on ProControl.

Page No	Control Number	Encoders	Units	Control Number	Switches	Units
1	1	Inp	dB	9	Inv (6)	On/Off
	2	Mix	%	10	Blank	
	3	Dly	ms/sec	11	Blank	
	4	FB	±%	12	FB/P	Pre/Post
	5	LPF	OFF/Hz	13	LP/P	Pre/Post
	6	BPM	bpm	14	Blank	
	7	Nte	*See below	15	Blank	
	8	Grv	%	16	Blank	

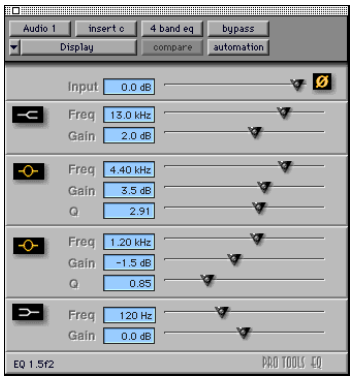
* Note values are: 1, 1+, 2, 2+, 4, 4+, 8, 8+, 16. These are not units -- instead, implemented with `GetValueString()`.

The format is very similar to the HUI Table explained above, except for the addition of symbols. Symbols are represented by numbers in the Encoders (PcTLCtls) and Switches (PcTL Sw) columns, and

are discussed in detail later in the chapter. Also of importance, ProControl does not have a VPOT Mode, as on the HUI.

ProControl requires parameter names to be no longer than three characters (which can include symbols). Note also, switch names can be up to four characters, meaning that three characters are then available for the state of the switch (allowing for a space between them). This works out well, since the state of a switch is often On or Off, or In or Out. In some cases however, abbreviations will be required; for instance Post in the ProX table above will be shortened to Pos.

More Page Table Examples with the 4-Band EQ



Mackie HUI Page Tables for 4-Band EQ

Page No	Switches	Units	Encoders	Units	VPOT Mode
1	Blank		Freq	Hz/kHz	0
	LoSh	In/Out	Gain	dB	1
	Blank		Freq	Hz/kHz	0
	HiSh	In/Out	Gain	dB	1
2	Inv	On/Off	Inpt	dB	2
	Blank		Q		3
	Blank		Freq	Hz/kHz	0
	HMF	In/Out	Gain	dB	1
3	Inv (Dupl)		Inpt (Dupl)		2
	Blank		Q		3
	Blank		Freq	Hz/kHz	0
	LMF	In/Out	Gain	dB	1

ProControl Page Tables for 4-Band EQ

Page No	Encoders	Units	Switches	Units
Page 1	Inp	dB	6	On/Off
	4,9 (ASCII SYM)	dB	HiSh	In/Out
	(4)"Fr"	Hz/kHz	Blank	
	"HM"(9)	dB	HiPk	In/Out
	HMF	Hz/kHz	Blank	
	HMQ		Blank	
	Blank		Blank	
	Blank		Blank	
Page 2	Inp (Dupl)		6 (Dupl)	
	"LM"(9)	dB	LoPk	In/Out
	LMF	Hz/kHz	Blank	
	LMQ		Blank	
	3,9 (ASCII SYM)	dB	LoSh	In/Out
	(3)"Fr"	Hz/kHz	Blank	
	Blank		Blank	
	Blank		Blank	

Feedback for Consistency!

Digidesign will work with developers by previewing plug-ins for a consistent paging scheme in relation to the full line of plug-ins for Pro Tools.

It is up to each plug-in developer to review and approve their own Page Tables. We always welcome betas, but can not guarantee that we will have time to evaluate their functionality (including Page Tables). If necessary, we can recommend nearby facilities with Digidesign control surfaces (eg Procontrol, Control|24) to help test Page Tables.

In addition, between the graphical interface of PeTE and the hidden menu in Pro Tools, the Digidesign Developer website also contains emulators for a few of Digidesign's control surfaces. We also encourage you to take advantage of the beta testers listed on the Developer website. We are confident that these tools will help your page tables be well conceived and useful for your users.

Alphanumeric Displays

For developers using the Control Manager, note that you do not have to explicitly handle the MapControlValToString() and similar calls - it does this for you!

With the advent of the newer advanced control surfaces (CS), plug-ins now have the opportunity to provide information to the user via alphanumeric displays on the CS. Unfortunately, the displays are limited in the number of characters that can be shown. For instance, on the Mackie HUI, nine characters are provided for each plug-in control, allowing only four characters for the control name, and four characters for the control value, and one for a space between them. On ProControl, eight characters are provided for each plug-in control, with three characters allocated to displaying a control name, and four characters for its control value. On Control|24 and Digi002, four characters are provided for the plug-in name, control name, and the control value. For the control value, these four characters include a +/- and/or any necessary unit abbreviations (ex: K, s, dB, etc.). D-Control and Command|8 provide six characters for the plug-in name, control name, and control value.

In order to display meaningful information in these short character strings, plug-ins will have to optimize the strings that they return to DAE. The dispatcher calls `MapControlValToString()`, which in turn calls `GetValueString()`, is used to obtain these control value strings. Typically in the past, the requested length argument of both these member functions, i.e., `maxChars` in `MapControlValToString()` or `maxLength` in `GetValueString()` has been ignored; but, from here on out, they should be carefully examined and used to create an optimum and meaningful string for any requested length.

To prevent the code from becoming unwieldy, as in the case of trying to provide strings for all requested lengths, a minimum number of expected lengths should be specifically addressed. The expected value lengths are: 4, 5, 6, 7, 8, and 31. In addition, ProControl switch states are a maximum length of 3 (i.e., when dealing with the state of a switch for ProControl, provide this information in 3 characters or less). Therefore, whenever possible, a plug-in should return the most meaningful string that will fit in any particular requested length, and at minimum, handle the expected lengths. If a plug-in does not have custom code to handle a particular requested length, it can round the length down to the next smaller expected length and use the code that it has for it. For example, a request of 9 characters should be converted to an expected request of 8 characters.

Please note: Since truncating a long string to fit within the requested length will not provide meaningful results in most cases, plug-ins must specifically provide code for deriving useful strings for the expected lengths at minimum — being as sophisticated as possible, when appropriate.

Since the smaller lengths, especially 4 and 5 characters, are usually too short to display a plug-in's true full value including units, some decisions will have to be made about how to suitably shorten them. The following provides some general guidelines.

If needed, and in order of precedence, try to:

- Remove spaces: `13 Hz` becomes `13Hz`
- Use common abbreviations for units: `16 seconds` to `16sec`, or `16 s`, `156 Hertz` to `156Hz`
- Drop the units entirely: `1832 Hz` to `1832`
- Round the value: `173.3 ms` to `173`

Here is a table depicting a typical example in more detail. The expected lengths are shown in the left most vertical column.

	1	2	3	4	5	6	7	8	9	10	11	12	13	....	31
31	1	5	6	.	2		H	e	r	t	z				
8	1	5	6	.	2		H	z							
7	1	5	6	.	2	H	z								
6	1	5	6	.	2										
5	1	5	6	.	2										
4	1	5	6												

While creating the above strings, the requested length argument passed in (`maxChars` in `MapControlValToString()`, and `maxLength` in `GetValueString()`) should be strictly adhered to! The string returned should be no greater than the requested number of characters. Furthermore, the developer should assume that the buffer passed into this function is only as large as the requested length. Any intermediate string processing should be done in temporary local buffers and only when you have the final string should you copy back to the buffer that was passed in, making sure you copy no more than the requested number of characters, plus the Null character or Length byte, as appropriate; since in general, `MapControlValToString()` uses C strings and `GetValueString()` uses Pascal strings.

Also, in an effort to further help prevent buffer overruns, two new functions have been added to the PI library file `SliderConversions.cp`: `SmartAppendNum()` and `SmartAppendXNum()`, which includes a maximum length argument and should be used in place of `FicAppendNum()` and `FicAppendXNum()` from `FicBasics.cpp`.

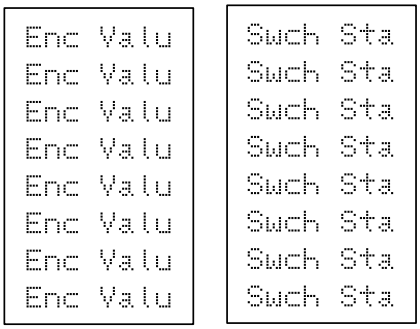
Finally, note that ProControl and HUI will sometimes utilize 5 characters to display its value when a sign is involved. For instance, if the number is -100%, the negative sign will appear in the space separating the control name from the control value. Pro Tools will take care of this conversion as long as all of the expected lengths are properly provided for. For Control|24 and Digi002, this is not the case - only four characters are allowed, so the +/- must be a part of those four characters.

- HUI, ProControl, Control|24, and Digi002 all provide alphanumeric displays for visual feedback, with the primary purpose being plug-in parameter editing. Functions in the plug-in library are provided that allow customized parameter strings to be created for use on the display. Specifically, these functions are: `GetControlNameOfLength()` and `GetValueString()`. Fortunately, the details of writing to the display are taken care of by the application. Therefore, the plug-in developer only needs to be concerned with providing meaningful display strings for all plug-in parameters that are controllable. Whether using the XML or legacy page table system, `GetControlNameOfLength()` should return the long version (31 characters maximum) of the plug-in's control names. Short versions of plug-in control names are stored in the XML file, edited with the PeTE application. If you are also using legacy page tables to support versions of Pro Tools prior to 6.4, the short names should also be coded in the `GetControlNameOfLength() function.`

DAE clients like Pro Tools call `GetControlNameOfLength()`, but if DAE finds XML data stored in the plug-in, it gets the information from there rather than calling into the plug-in. `GetControlNameOfLength()` is responsible for providing the parameter "names" used on the display. As with parameter "value" strings, it is important to carefully create parameter names that are meaningful in the limited space allocated to displaying parameter names. On ProControl, string lengths of three characters will be used for the parameter name strings, where four characters will be used on the HUI, Control|24, and Digi002. D-Control and Command|8 use six characters for control names. In general, lengths of 3, 4, 5, 6, 7, 8, and 31 should be specifically addressed in the PeTE editor or `GetControlNameOfLength()`. We begin next, by looking at some concrete examples.

The ProControl Display

ProControl also provides an alphanumeric display; however, there are some significant differences that need to be addressed. Shown is a generic representation for the ProControl display. The left pane is what users will usually be viewing on ProControl's displays, which are the current Encoder/Value settings. The right pane shows the current switch settings when the user toggles to the Switch/State display mode. ProControl will only display one of these views at any given time.



As with the HUI, meaningful strings will have to be provided in the limited available lengths. ProControl, Control|24 or Digi002 value strings require no special treatment other than what has already been stated above for HUI, since all of these control surfaces display their values in 4 digits/characters, as returned by `GetValueString()`. The biggest difference between the HUI display and the Digidesign control surfaces displays is that Digidesign CS's can display symbols.

Let's take a moment to explain how the displays of Control|24 and Digi002 work. Both of these surfaces have a four character LED display located above each encoder/switch pair. When in Channel mode, these scribble strips show the plug-in name (if any) on that channel. When that plug-in is selected, the display switches to show the controls for the plug-in. Now each display shows the name of the control. The display automatically switches to the current value of the control when the encoder or switch is moved or pushed.

Digidesign Extended Character Set

As of this writing, this section applies to ProControl, Control|24 and Digi002

As seen, a maximum string length of 3 or 4 is limiting when attempting to display encoder parameter names, and in some cases switch states, as well. For instance, "Enc" may not be very informative! To help alleviate this, symbols are available with some Digidesign Control Surfaces to help the user interpret the encoder or switch function. The following enumerations are found in `FicHWController.h`. Depictions of the symbols are in the figure to the right.

1. `eDAEHWControllerSymbols_LowPassFilter`
2. `eDAEHWControllerSymbols_HighPassFilter`
3. `eDAEHWControllerSymbols_LowShelfFilter`
4. `eDAEHWControllerSymbols_HighShelfFilter`
5. `eDAEHWControllerSymbols_Peak`
6. `eDAEHWControllerSymbols_Phase`
7. `eDAEHWControllerSymbols_UpArrow`
8. `eDAEHWControllerSymbols_DownArrow`
9. `eDAEHWControllerSymbols_PlusMinus`
10. `eDAEHWControllerSymbols_LoPeak`
11. `eDAEHWControllerSymbols_HighPeak`
12. `eDAEHWControllerSymbols_LeftRight`
13. `eDAEHWControllerSymbols_Link`
14. `eDAEHWControllerSymbols_Compressor`
15. `eDAEHWControllerSymbols_Limiter`
16. `eDAEHWControllerSymbols_Auto`
17. `eDAEHWControllerSymbols_Key`
18. `eDAEHWControllerSymbols_KeyListen`

Symbols can be combined with text, using the following guidelines.

■ The combination of symbols and associated text must fit in a string length of 3 (ProControl) or 4 (Control|24 and Digi002) characters maximum. Take, for example, a string comprised of two symbols and a single text character: "`L[11][9]`", where "L" refers to "Left", [11] = High Peak symbol and [9] = Plus/Minus symbol. All characters will necessarily have to be placed one after the other in the string, so do not exceed 3 or 4 characters. In PeTE, [11] would be replaced by the token: <!HiPk!>. See the Name Editor in PeTE and its documentation for more information.

■ If a string is made up of a single text character and a single symbol, e.g. "`L [1]`", where "L" refers to "Left" and 1 = LowPassFilter symbol, there should be a space delimiter between the text character and the symbol. Again, in PeTE the list of tokens is shown in the Name Editor.

■ Note that any string shorter than 3 or 4 characters will automatically be left-justified on the display by the application.

Note: There is still room for defining additional symbols if you have a plug-in that can benefit from it. If you think you have a symbol that may prove useful, please contact us so that it can be considered for inclusion. Email: devservices@digidesign.com

The Mackie HUI Display

The Mackie HUI has a Vacuum-Florescent Display (VFD) that provides alphanumeric information to the user. Shown here is the general graphic template for the HUI VFD, including definitions for the abbreviations used. The template illustrates the characteristic plug-in parameter format for this display type.

Swch = Switch - parameter name assigned to the switch (4 chars maximum)
Stat = State - current state of the switch parameter (4 chars maximum)
Encr = Encoder- parameter name assigned to the encoder (4 chars maximum)
Valu = Value - current value of the encoder parameter (4 digits maximum)

Vertical white lines delineate the separation of fields, though not actually shown on the HUI VFD. The critical thing to notice here is the short 4 character strings that are required for all of the parameters! It is up to the plug-in developer to provide meaningful 4 character parameter names, for the parameter name assigned to the switch and the parameter name assigned to the encoder, and to provide meaningful 4 character parameter values in these strings.

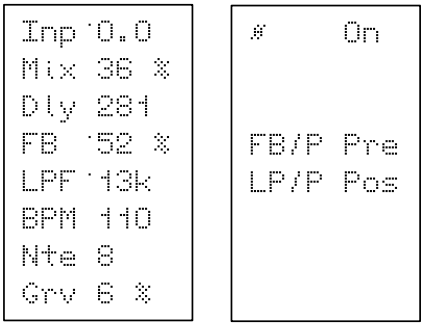
Note, in the above "Valu" field, if the value is negative, the negative sign will be displayed in the space between the "Encr" and the "Valu." Therefore, negative values are still shown using 4 significant digits, i.e., negative values will have a string length of 5. These details are taken care of by the application as long as proper length strings are provided for in your plug-in.

Next we discuss two specific examples: the ProX and 4-Band EQ plug-ins. Some typical output on the HUI VFD is shown, along with references to code examples for display strings.

ProX on the HUI and ProControl Displays

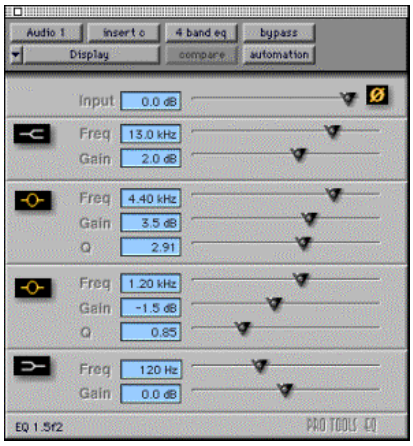
Display output representing the current settings on the ProX plug-in is depicted in the following display images. Notice, it requires two pages to map all of the controls on the ProX plug-in to the HUI. Note also, that page 1 of the HUI VFD still includes the white vertical separators for reference; though these are dropped in all subsequent HUI displays, since on the real HUI the switches and encoders provide a visual reference for the separation of fields.

ProControl Displays



HUI Displays

4-Band EQ Displays



Mackie HUI 4-Band EQ Displays

```
          LoSh Out                HiSh Out
Freq 120  Gain 0.0  Freq 13.0 Gain 2.0
```

```
Inv  On                         HMF  In
Inpt 0.0  Q    2.91 Freq 4.40 Gain 3.5
```

```
Inv  On                         LMF  In
Inpt 0.0  Q    0.85 Freq 1.20 Gain-1.5
```

As may not be apparent at first, the HUI display lends itself well to certain types of parameter placement and naming conventions, which sometimes may lead to a clearer description of the parameters. The 4-BandEQ plug-in is a good example of this. On the HUI displays, you will notice that there are four parameters labeled "Freq" and four parameters labeled "Gain." How is the user supposed to know which "Freq" or "Gain" control belongs to which band? Why not let the switches convey this additional information! By judicious placement and naming of switches, the "Freq" and "Gain" controls become grouped with the corresponding switch name. When this method can be applied, it helps to make what would otherwise be cryptic parameter names, less cluttered. For instance, "Freq" is clearer than "LSFq" when referring to "Low Shelf Frequency," as long as it's understood that the "Freq" control, will adjust the low shelf frequency.

You can take advantage of this in `GetControlNameOfLength()` by checking if the control surface type being passed in is the HUI (i.e., page table type: `cMackiePageTable`). If your plug-in lends itself well to this type of layout, return appropriate control names tailored to this format.

While PeTE will import data from legacy plug-ins that have used the above tactic, it will only save new names in either the Digidesign Extended character set, or ASCII set. The extended set applies only to control surfaces that support it, and ASCII will apply to all others.

ProControl 4-Band EQ Displays

```
Inp 0.0
< ± 2.0
<Fr 13.0
HM± 3.5
HMF 4.40
HMQ 2.91
```

```
#    On
HiSh Out

HiPk In
```

```
Inp 0.0
LM± -1.5
LMF 1.20
LMQ 0.85
> ± 0.0
>Fr 120
```

```
#    On
LoPk In


LoSh Out
```

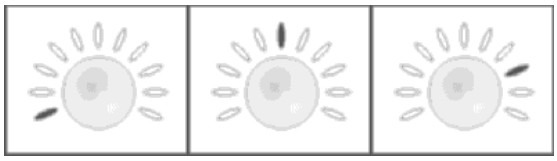
Rotary Encoder-LED Displays

Some advanced control surfaces, such as HUI, Control|24 and Digi002, have a set of radial LEDs surrounding each encoder. The LEDs provide a visual cue of the current value of the control to the user. Moreover, there are several "modes" of operation that allow the LEDs to also convey additional information. These modes are discussed here, along with how to set them. Note that in the encoder-display images below, red signifies an illuminated LED segment.

Single Dot Mode

For new developers using the Control Manager, GetControlOrientation() is handled for you. Most likely, you need not be concerned with setting the orientation. For example, using a CPlugInControl_BoostCut appropriately sets the boost/cut mode. Simply read this section with the pleasure of knowing how the pretty lights work, but not having to worry about it!

The first mode is *single dot* mode. This is the standard for all controls that do not fall into any of the special cases presented next. In this mode, only one LED segment is lit at a time — traveling from the minimum to the maximum value as the control is rotated throughout its range. Note that the minimum or maximum "orientation" is programmable; with minimum (or maximum) either appearing on the left or the right. To set the orientation for minimum on the left and maximum on the right, return `kDAE_RotaryLeftMinRightMax` in `GetControlOrientation()`. Otherwise, for minimum on the right, and maximum on the left, return `kDAE_RotaryRightMinLeftMax`.

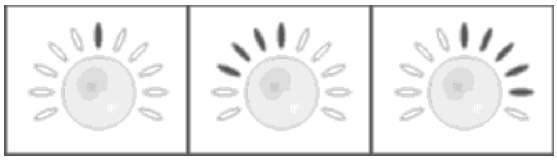


Returning `kDAE_RotarySingleDotMode` in `GetControlOrientation()` will set the LED encoder-display to *single dot* mode.

The next two modes should only be used with gain and level controls that are expressed in units of dB! This is a standard that we will adhere to — so that the user can trust that he or she is working with a dB level when viewing one of these LED display modes. Typical examples are: EQ boost/cut, input level, send level, etc. Your plug-in should always follow this standard.

Boost/Cut Mode

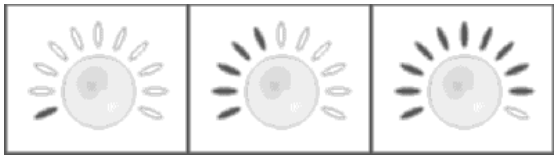
The first of these specialized modes is *boost/cut* mode. As its name implies, this is to be used for the boost/cut gain of an EQ plug-in. It has the characteristic of being 0 dB at the 12 o'clock position, and either illuminating LEDs to the Left (for cut), or illuminating them to the right (for boost). For example, the leftmost image shows a unity gain of 0 dB. The middle image shows a cut in the gain level, while the rightmost image shows a boost in the gain level. Note that it is possible to change the orientation as mentioned previously, but may not make much sense when using *boost/cut* mode.



Returning `kDAE_RotaryBoostCutMode` in `GetControlOrientation()` will set the LED encoder-display to *boost/cut* mode.

Wrap Mode

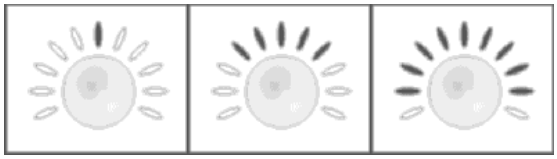
All other input and output levels that are expressed in dB, excluding *boost/cut* mode, should use *wrap* mode. *Wrap* mode incrementally lights the LEDs as the encoder is rotated throughout its range to display the dB value. Typical examples would be: input levels, send levels, or any other gain or level control that can be expressed in dB, except for *boost/cut*. As with *single dot* mode, the minimum or maximum orientation of *wrap* mode is programmable; with minimum (or maximum) either appearing on the left or the right. To set the orientation for minimum on the left and maximum on the right, return `kDAE_RotaryLeftMinRightMax` in `GetControlOrientation()`. Otherwise, for minimum on the right, and maximum on the left, return `kDAE_RotaryRightMinLeftMax` in `GetControlOrientation()`.



Returning `kDAE_RotaryWrapMode` in `GetControlOrientation()` will set the LED encoder-display to *wrap* mode.

Spread Mode

Finally, the last mode is a unique case, called *spread* mode. It is only to be used for the "Q" control of an EQ plug-in. It has the following general characteristics: for high Q (i.e., narrow bandwidth), it will appear as in the left most image below (as a single illuminated LED at 12 o'clock). For low Q (i.e., wide bandwidth), the illuminated LED segments will spread out more, much in the same way that the bandwidth spreads out. The two images on the right depict the Q setting as it becomes more and more broad. Again, it is possible to change the orientation as mentioned previously, but does not make sense to do so when using *spread* mode.



Returning `kDAE_RotarySpreadMode` in `GetControlOrientation()` will set the LED encoder-display to *spread* mode.

Encoders and Control Modes

There are two types of control modes for rotary encoders: Normal and Fine. Normal Control Mode is the default behavior of moving the knob. Fine Control Mode is achieved by moving the knob while the surface's modifier key is held down.

In Normal Control Mode, One complete turn of the control surface knob will move the control through the full range of values between min and max (as defined in the control's constructor), but it will not necessarily increment through every value possible. Some values will be skipped due to a larger increment size. This larger increment size is dependent on the control surface implementation. For example, the 002's physical knobs have about 25 discrete clicks in one turn. So the plug-in control will assume ~25 different values between min and max during one knob rotation.

Fine Control Mode, on the other hand, will increment/decrement the control's value by the `stepSize` (defined in the control's constructor), incrementing through each possible value. Each click represents 1 `stepSize` in difference.

There is no direct relationship between normal and fine control step size. It is control surface dependent. The plug-in has no way to specify how a control surface knob will affect a control in Normal Control Mode, other than specifying the min and max values.

Encoders and Small Relative Token Changes

This section is irrelevant for new developers making use of the Control Manager.

It is important when storing continuous control values internally in your plug-in, to do so with adequate step resolution. If this is not done, it is possible that relative change tokens (generated by slowly rotating an encoder on a hardware control surface) may be missed; resulting in no change of the parameter! This could happen if these small (delta) relative change tokens are not accumulated, due to internal rounding by the plug-in.

For example, if DAE increments a control's value by sending out a relative change token of 1 (a very small amount, considering a 4-byte long — or 32-bit value), the internal representation of that control in the plug-in should also be incremented by 1. Moreover, if DAE immediately requests this same value back, it should get back the control's value incremented by 1! If it does not, internal rounding by the plug-in has caused a loss in accuracy, and this same rounding effect can ultimately cause encoders to be nonfunctional.

Ideally, the correct way to circumvent this problem is to store continuous control values as 4-byte longs. This is the same data type used by DAE; therefore, any values passed back and forth by the token system to and from the plug-in will not have reduced accuracy. Moreover, when both DAE and plug-ins use the same data type, everything functions as designed.

Plug-in developers should begin to adopt the standard practice of using4-byte long data types (or 32-bit values) for storing all continuous control values.

Since it is conceivable that many plug-ins do not currently use longs for control value storage, large overhead could be incurred by rewriting code to this specification. Therefore, in the meantime, it is possible to examine each continuous control separately to determine if it might have this problem. A general rule is that if you are storing a continuous control with “coarse” resolution (i.e., approximately 75 discrete steps or less for its full range), you should change it to a long, and quantize in places where you need to use the coarse value. Since the number 75 is empirically derived with current controllers, this value will change as the resolution on hardware controller changes (or any external control software module); hence the need for the robust solution described here (i.e., using long data types).

There is another possibility of keeping track of accumulated values so that controls do get updated when small relative change tokens are generated— though the general idea is only briefly described next, since the preferable, and long term solution (no pun intended), is to store these values as longs:

If your continuous control is quantized with coarse step resolution, you need a way to keep track of any small relative change tokens that are sent to `UpdateControlValueInAlgorithm()`. Rather than discard them (e.g., if they do not increment/decrement your control into the next bin, during an update of the control value), they should be accumulated so that the control will eventually increment/decrement as appropriate (after enough of them have been sent). (Note that

`CProcess::UpdateControlRelative()` is responsible for dispatching these relative tokens, which are then sent to `UpdateControlValue()`.)

Note again that this refers to continuous controls. Discrete controls will not exhibit this behavior, since the plug-in library already has provisions to correctly implement discrete controls.

Encoders and Full-Off/Full-On Values

Again, this section is likely irrelevant for new developers making use of the Control Manager.

Another important detail of DAE's internal representation of continuous controls concerns the full-off/full-on value of encoders and their corresponding LED tick marks (i.e., the radial LED segments surrounding each encoder). Both the minimum and maximum LED tick marks are affected by DAE values that represent full-off and full-on. Note that DAE always uses `0x80000000`/`0x7FFFFFFF` to represent full-off/full-on, respectively (unless reversed, using `GetControlOrientation()`). Therefore, in order to change the illuminated state of either the maximum or minimum LED segment, exact DAE values of `0x80000000` or `0x7FFFFFFF` must be used! If a plug-in's parameter is changing value from full-off/full-on, it will be necessary to change the illuminated state of the minimum/maximum LED; which is done by using values that differ from `0x80000000`/`0x7FFFFFFF`; resulting in a change of state of the minimum/maximum LED segment, respectively (with normal orientation). In other words, `0x80000000` changed to `0x80000001` will result in a change of state of its corresponding, outermost, LED, as will `0x7FFFFFFF` changed to `0x7FFFFFFE`.

In short, be sure that your plug-in returns these exact full-off and full-on values at the extreme of the controls range, and returns values that differ from these when it has been adjusted off of the minimum or maximum setting. Again, this only concerns continuous controls, since the plug-in library will handle discrete controls properly. Do note, however, that if your continuous control is quantized (as are most controls), and in effect — controls a parameter discretely; a range of values falling within any full-off or full-on bin, will all have to be mapped to either `0x80000000` or `0x7FFFFFFF`, to keep the illuminated state of the LED segment accurate. Conveniently, there are routines in `SliderConversions.cpp` in the plug-in library that can be used to convert between 32-bit DAE values and the actual control values used in a plug-in.

Other Advanced Control Surface Features

This section provides information on features that may be included on advanced control surfaces that need consideration for plug-in development. As of this writing, Digidesign control surfaces - ProControl, Control|24, Digi002 Command|8, and D-Control - are the only control surface utilizing the features mentioned below.

Plug-in Categories

ProControl, Control|24, Digi002, Command|8, and D-Control have dedicated EQ and Dynamics switches allowing quick access to any available EQ or Dynamics plug-ins. These same switches can also be used to toggle the bypass state of an EQ or Dynamics plug-in. Digi002 has one global EQ and one global Dynamics switch. When one of these switches is pressed, the alphanumeric display for the channel strip containing the most recently accessed EQ or Dynamics plug-in begins to flash. To implement this, these control surfaces must be able to determine the types of plug-ins currently available. See the "Plug-In Categories" section in the **Plug-In Features** chapter for information on how to properly report your plug-in's category.

Plug-Ins With Multiple Categories

If your plug-in has multiple sub-sections, e.g. EQ and Dynamics, you should override `CProcessType:GetFirstPageForCategory()`. In `CYourProcessType::GetFirstPageForCategory()` you should return a page number based upon controller type and category. In practice, this allows ProControl to go to the first page of the sub-section. If your plug-in is a single category, page 1 is automatically returned for you.

In addition, for advanced control surfaces to individually bypass each sub-section on your plug-in, you may need to override `CProcessType:GetIsBypassableByCategory`, along with `CProcess::GetBypassControlByCategory()` to specify the bypass controls.

In `CYourProcessType::GetIsBypassableByCategory()` you should return bits set for all the categories that can bypass. If your plug-in is a single category, and has a master bypass, this is automatically done for you. If `GetIsBypassableByCategory()` is not overridden, it is assumed that all sub-sections are bypass-able, as returned by `Gestalt(pluginGestalt_CategoryBits)`.

In `CYourProcess::GetBypassControlByCategory()` you should return a bypass control number that corresponds to the category specified. For example, a combo EQ and Dynamics plug-in would return the bypass control for its EQ section when `ePlugInCategory_EQ` was passed into the routine.

D-Control Emulator

Once you've finished using PeTE to create your plug-in's page tables for the D-Control it is a good idea to test to make sure they work. The D-Control emulator, which emulates the real D-Control unit in software, makes this possible.

Requirements

Unlike the Procontrol emulator, the D-Control emulator runs only on Mac OS X. You can use it, however, to test a plug-in running in Pro Tools on either a Mac or a remote PC.

The requirements for testing on the Mac are as follows:

- A Mac running OS X 10.2.x or higher.
- A debug build of PT 6.4 TDM (or higher) for Mac. PT LE will not work.

The requirements for testing on the PC require the above, plus the following:

- A separate PC running Windows XP with an available Ethernet connection adapter in the CPU.
- A debug build of PT 6.4 TDM (or higher) for Windows.
- Both Mac and PC connected to the same ethernet LAN.

Setting Up the Emulator

The D-Control emulator can be found on the developer website in the 'Utilities' section. Here are the steps necessary to get the emulator working:

1 Start by installing a debug build of Pro Tools 6.4 (or higher) for the Mac. If you are testing on a Mac you can run Pro Tools on the same machine that the emulator is running on. Copy the emulator, PC2_Emulator_OSX into the Pro Tools debug directory. If you are testing on a PC you must additionally install a debug build of PT 6.4 (or higher) on your Windows machine. Please follow the instructions included with the download for installing debug builds.

2 Install the Digi ethernet protocol on your Windows XP machine (this step is only necessary if you are running PT on a remote PC):

- In the 'Network Connections' Control Panel, double-click on 'Local Area Connection'.
- In the 'General' tab, click on the 'Properties' button. The 'Local Area Connection Properties' window should appear.
- In the 'General' tab, click on the 'Install...' button.
- Select 'Protocol' in the list of options, and click on 'Add...'.
- In the 'Select Network Protocol' window that appears, click on the 'Have Disk...' button.
- Click 'Browse...' and browse to `\Program Files\Common Files\Digidesign\DAE\Controllers\` and select the `DigiNet.inf` file.
- Click 'OK' and you should see "Digidesign Ethernet Support" appear in the 'Select Network Protocol' window. Click 'OK' again.
- The protocol should now be installed and you should see 'Digidesign Ethernet Support' property listed in the 'Local Area Connection Properties' window.
- Restart your machine after installing.

3 Launch the Emulator by double-clicking on `PC2_Emulator_OSX`. If you are having trouble launching, make sure you have not removed the `DigiMachOServices.framework or DigiShoeTool` from the folder – they must be present in the same folder as the emulator. Make sure also that `tcl.framework` is present,. It should be located in `/System/Library/Frameworks`. When you first launch the emulator you won't see much happening – just a menu appears. As you'll see in the next step, you'll need to open up some configuration files to do anything useful.

4 As with the real D-Control unit, the emulator is modular, and consists of one or more separate sections. These include a "center" section (which is the main unit) and various "fader pack" sections. To test your page table implementation you must open at least one center section and one fader pack. Which files you should use is dependent on whether you will be running Pro Tools either locally on the same Mac that you are running the emulator on, or remotely on a PC.

Local machine (Mac):

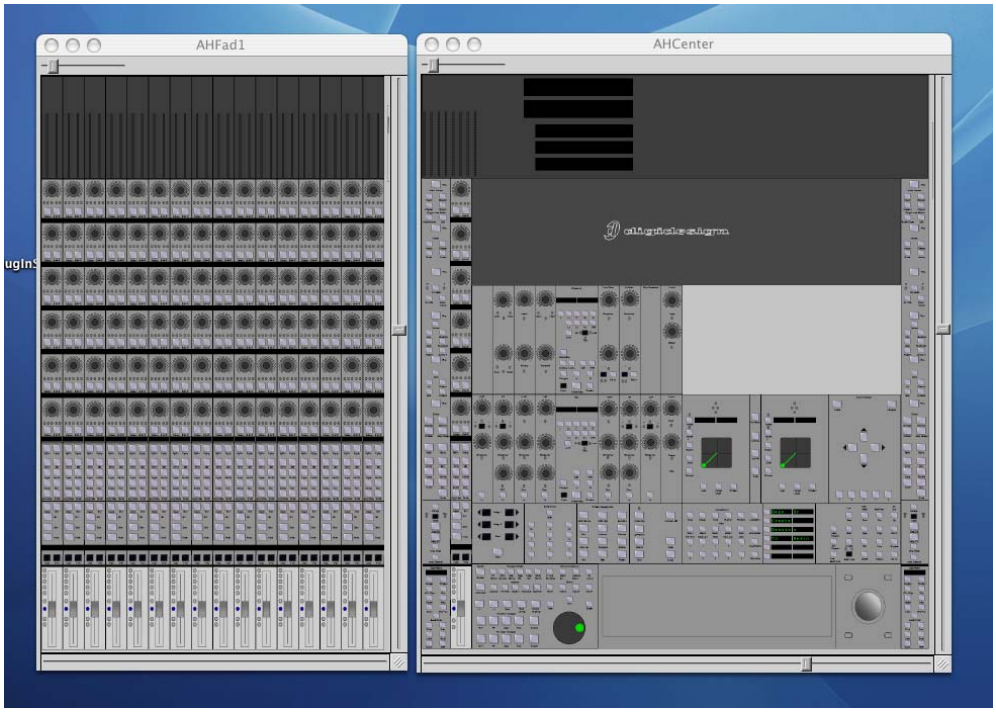
Open the "A" center section by going to "File" > "Open" and then selecting "AHCent". You should see the UI of the center section appear on your screen. Now open the "A" fader pack by going to "File" > "Open" and then selecting "AHFad1".

Remote machine (PC or Mac):

Open the "B" center section by going to "File" > "Open" and then selecting "B2Cent". You should see the UI of the center section appear on your screen. Now open the "B" fader pack by going to "File" > "Open" and then selecting "B2Fad1".

Note that you can't simply double-click on these configuration files – you must open them from within the emulator by using the File menu.

Once you have both sections open you should see something like this on your screen:



5 Now that you have the emulator up and running, the next step is to launch Pro Tools. Make sure you are using the debug build, and not the release build.

6 In Pro Tools you'll need to set up and enable interaction with the emulator. This works in exactly the same way as it would with the real hardware unit. Go to "Setups" > "Peripherals..." and select the "Ethernet Controllers" tab. Check the "Enable" check-box. When you do this, Pro Tools will scan the network for any connected ethernet controllers. Now "AHCent" and "AHFad1" (or "B2Cent" and "B2Fad1" if you're on the PC) should appear in the pull-down menus next to the colored boxes, which are labelled 1 – 7. Next to box #1 select "AHCent" (or "B2Cent"). Next to box #2 select "AHFad1" (or "B2Fad1"). Click "OK".

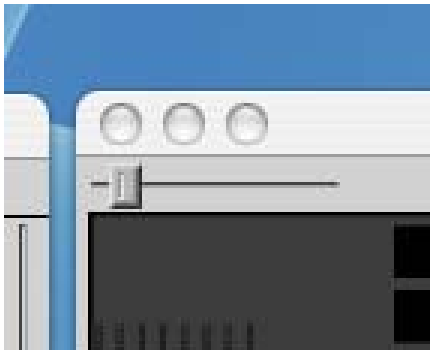
7 If asked to download the current version of the firmware, choose "Cancel".

8 If you haven't already done so, open up a session. The D-Control emulator UI should now reflect your opened session, and you should also have full control over Pro Tools from the emulator. The emulator should behave just like the real hardware unit.

Working with the Emulator

Although the emulator can take some time to learn, it is not hard to test plug-ins with it. It is, however, tricky to manipulate the UI of the emulator. A quick tutorial on zooming with the emulator:

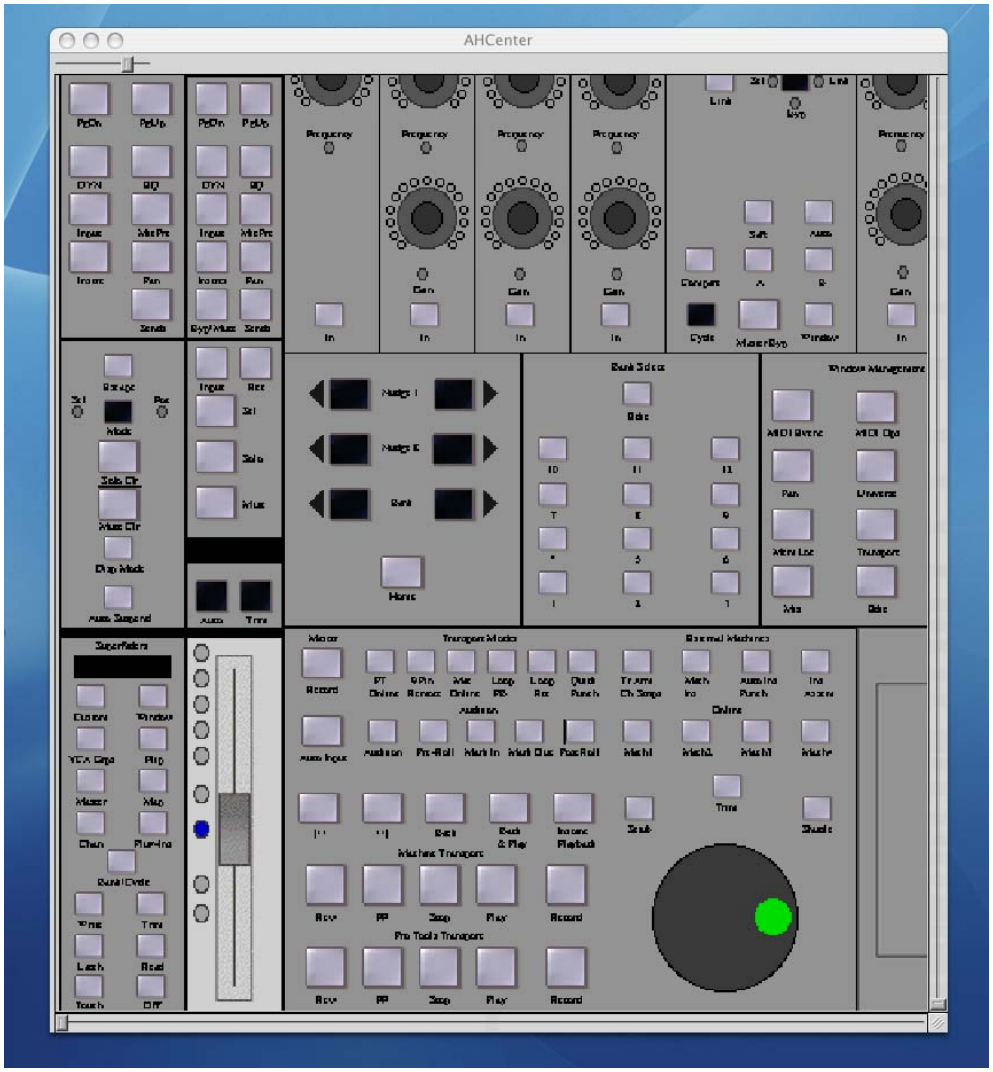
In the upper left corner of each unit you'll see a small slider:



This is the “zoom” slider and doesn’t operate quite like other zoom sliders you may have seen. It zooms in by first trying to expand the window containing the unit itself. If the window has reached the edge of your screen it will magnify within the (now) fixed window. If you have a second monitor attached to your machine you should not try to zoom while the unit is in the second monitor. It will try to expand the window to the edge of the *first* monitor, producing undesired results.

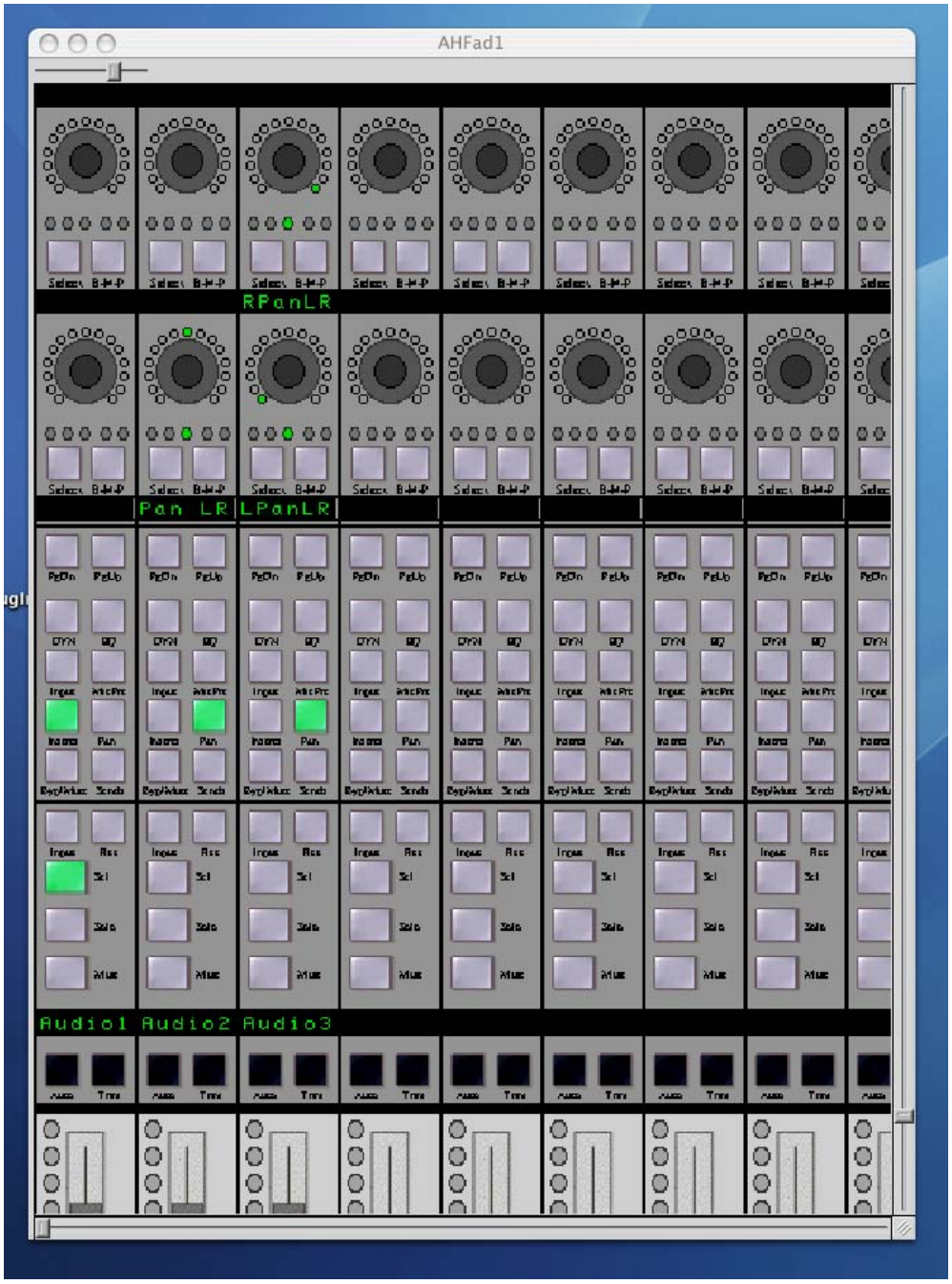
Once you’ve zoomed into the desired magnification, the next step is to navigate to the area of the control surface that you want to look at. You do this using two additional sliders located on the bottom and right sides. Unfortunately the graphics for these sliders do not update quite properly after a zoom (yes, this is a bug...) and must be clicked once or twice before they operate correctly.

Except for Dynamics or EQ plug-ins, for purposes of checking your page table implementation in D-Control you won’t care much about most areas of the main unit. Zoom into the lower-left corner of the main unit so that it looks something like this:



As you can see, even at large magnification it is nearly impossible to read the labels on the controls. To help get around this problem, we're including reference images that are clearly labelled in the instructions below that you can use to operate the emulator.

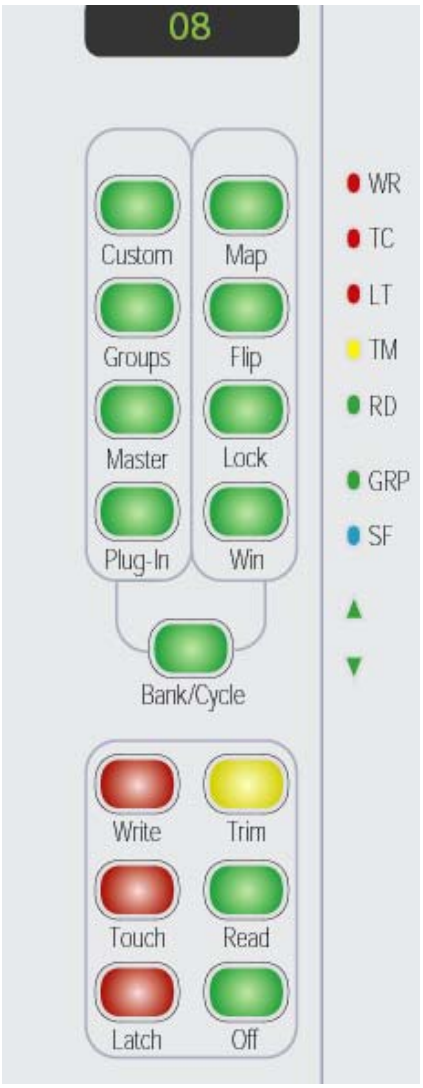
You'll need to navigate through a larger portion of the fader pack to do you page table work. Set up the fader pack to look something like the following, so that you have at least 8 channel strips showing:

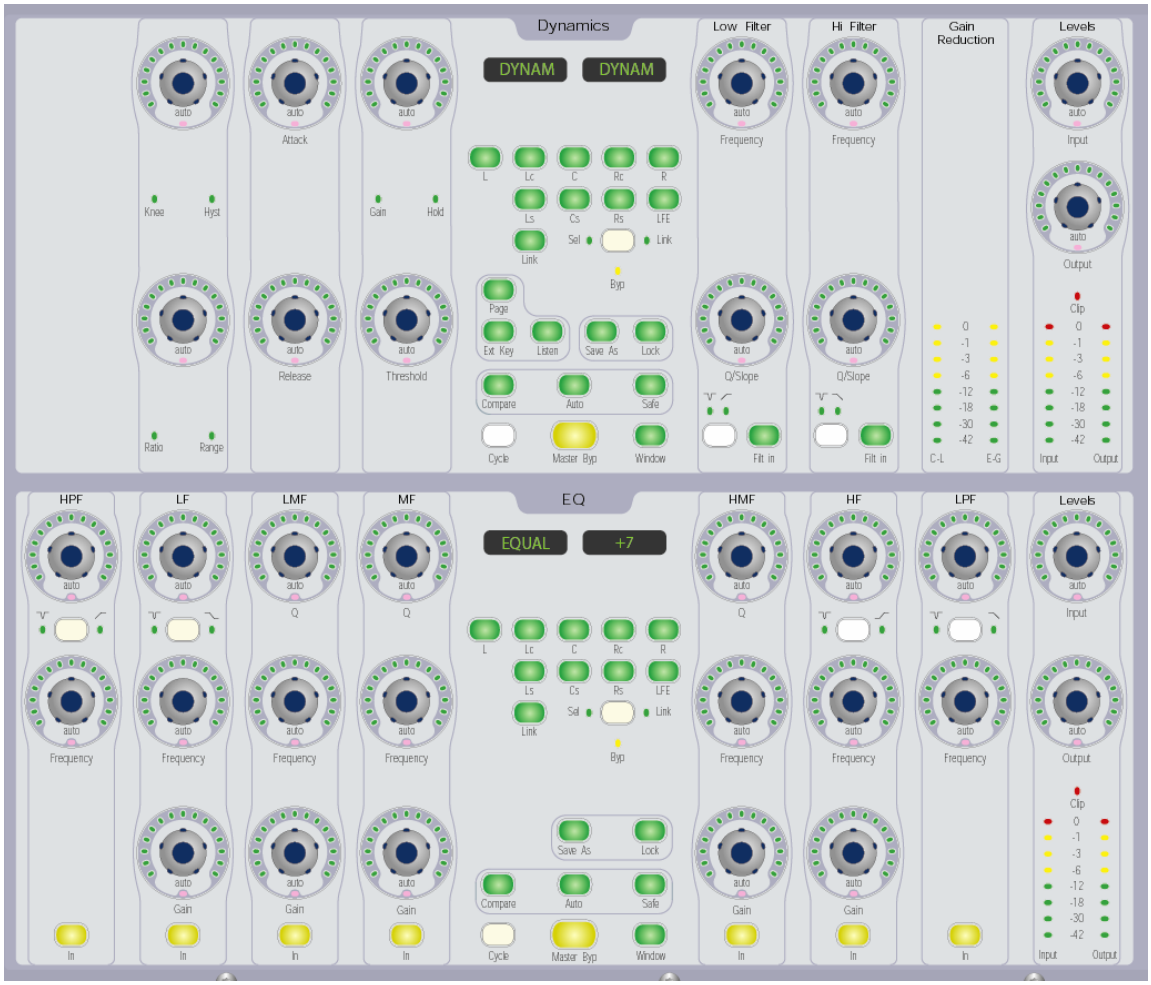


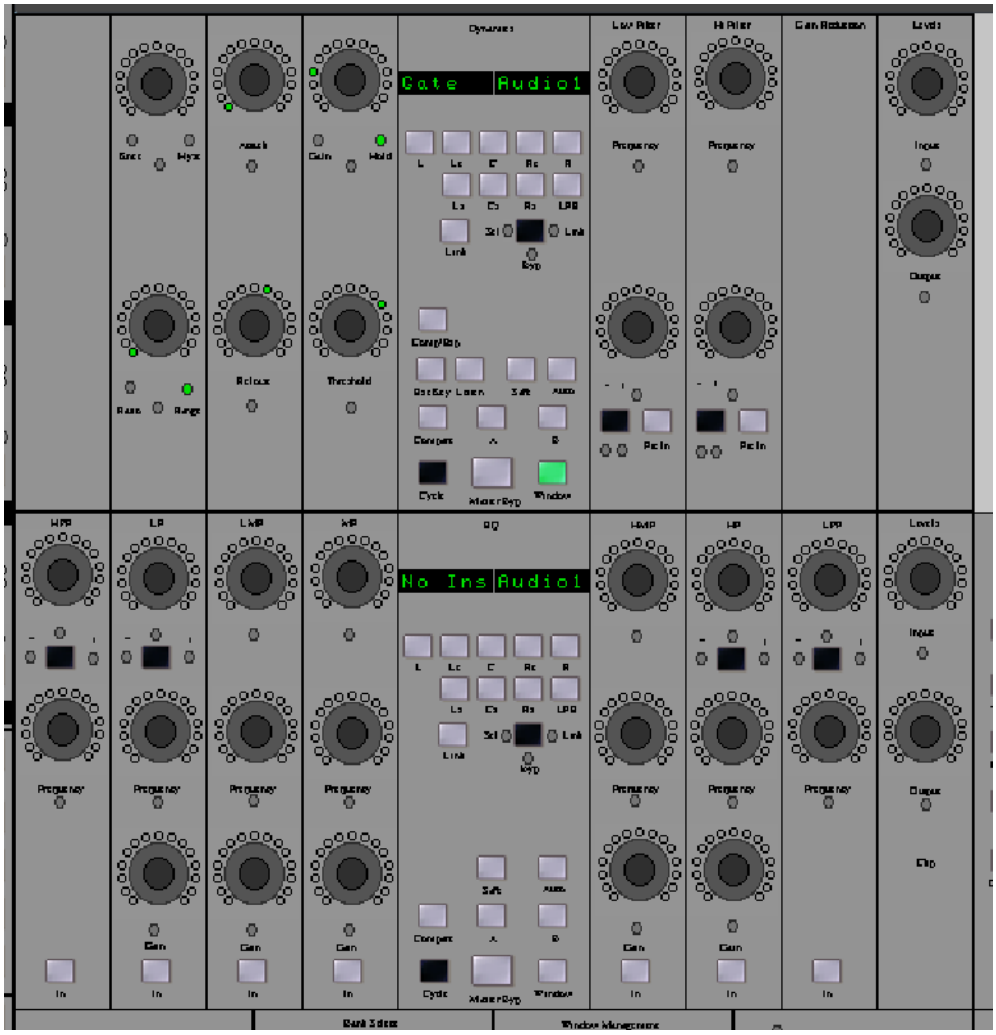
Scroll up and down using the scroll control on the right side to get a feel for how to quickly look at different parts of the channel strips.

Emulator Reference Images

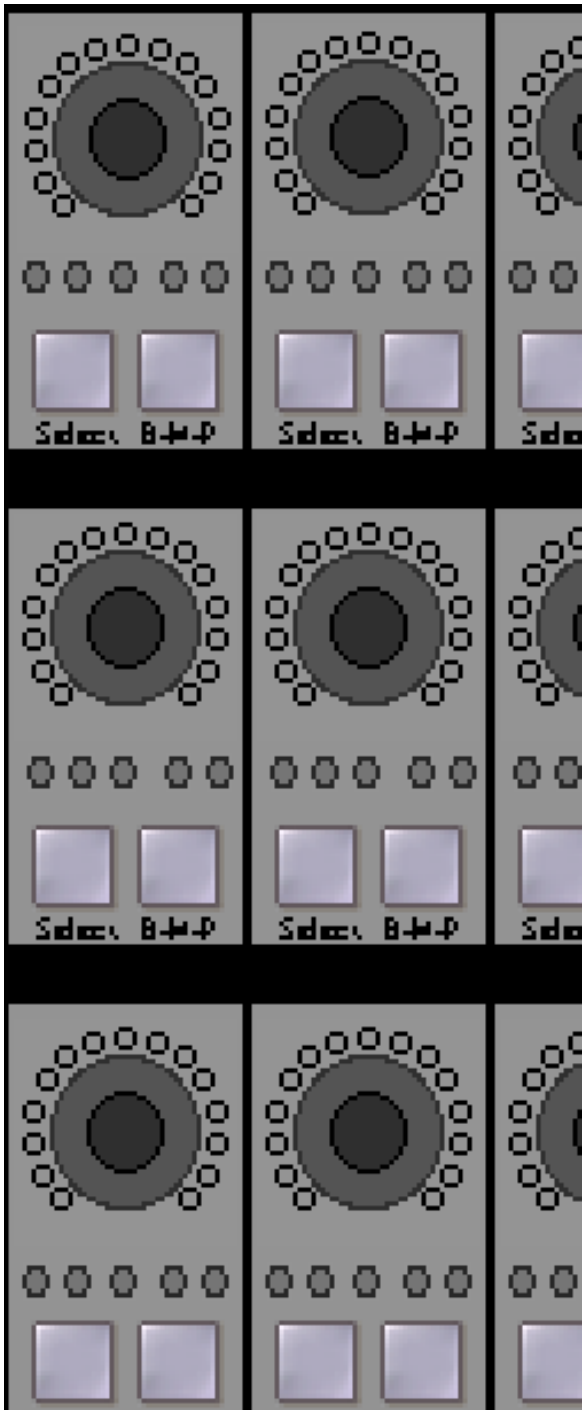
Use these side-by-side comparison images to help you overcome the resolution limitations of the emulator.




Dynamics
DYNAM
DYNAM
Low Filter
Hi Filter
Gain Reduction
Levels
Attack
Frequency
Input
Knee
Hyst
Gain
Hold
L
Lc
C
Rc
R
Ls
Cs
Rs
LFE
Link
Sel
Link
Byp
Output
Page
Clip
Release
Threshold
Ext Key
Listen
Save As
Lock
Q/Slope
Compare
Auto
Safe
Ratio
Range
Cycle
Master Byp
Window
Filt in
C-L
E-G
Input
Output
EQ
HPF
LF
LMF
MF
HMF
HF
LPF
Levels
EQUAL
+7
Q
Frequency
Gain
In


Gate
Audio1
No Ins
Audio1


auto
SEND A
Dyn Eq Mute Pre
Select B·M·P
auto
SEND A
Dyn Eq Mute Pre
Select B·M·P
auto
SEND B
Dyn Eq Mute Pre
Select B·M·P
auto
SEND B
Dyn Eq Mute Pre
Select B·M·P
auto
SEND C
Dyn Eq Mute Pre
auto
SEND C
Dyn Eq Mute Pre

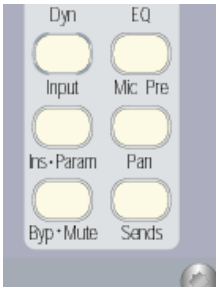



auto
auto
SEND F
SEND F
Dyn Eq Mute Pre
Dyn Eq Mute Pre
Select B·M·P
Select B·M·P
Page Dn Page Up
Page Dn Page Up
Dyn EQ
Dyn EQ
Input Mic Pre
Input Mic Pre
Ins·Param Pan
Ins·Param Pan
Byp·Mute Sends
Byp·Mute Sends
Auto/Inpt Record
Auto/Inpt Record
Sel
Sel
Solo
Solo
Mute
Mute
KICK
KICK
Auto Trim
Auto Trim
Pan LR
Audio1
Audio2

Checking Plug-Ins with the Emulator

"Channel Strip" Mode

This is the standard way that users will use D-Control to control your plug-in.



1 Insert your plug-in in the first insert location on the first track in your session in Pro Tools.

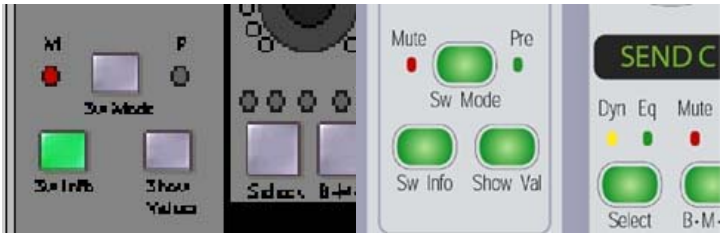
2 In the corresponding first channel strip in the "fader pack" section of the emulator, click on the 'Ins/Param' button, next to the 'Pan' button. This will set the channel strip in insert mode.

3 Scroll up to the top of the channel strip, and you should see the name of your plug-in in the scribble strip. Click on the 'Select' button, and you should see the names of your plug-in controls instantly laid out on the channel strip.

4 You should now be able to move through and test the full range of your controls, just as you would on the real surface. Note that in order to test "Fine Control Mode" of a control, you must hold the Cmd (on Mac) or Ctrl (on Windows) key on the machine that is running Pro Tools, even if Pro Tools is running on a different machine from the emulator, and then move the encoder in the emulator.

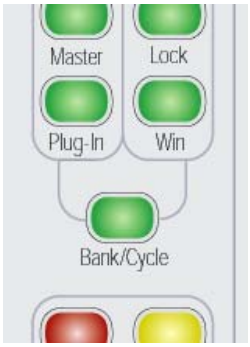
5 If you have more than one page of controls you can flip through the pages on the emulator by clicking on the 'Page Up' and 'Page Dn' buttons which are located at the top of the bottom panel on the channel strip.

6 The names of controls placed on switches will not appear in the scribble strip unless the Sw Info button is pressed. The Sw Info button is located in the Center Section (AHCent or B2Cent) toward the upper left corner. See the pictures to the right for reference. The presence of a control on a switch, however, will be indicated in the Channel Strip by the far-right 'Pre' light lighting up. Actuate a discrete control by clicking on the 'B•M•P' button.



"Custom Fader" Mode

This mode allows a much expanded view of the plug-in, with up to 48 encoder / switch pairs per page.



1 Follow the directions above for editing in "Channel Strip" mode.

2 In the center section of the emulator, click on the 'Plug-In' button, which is to the above-left of the 'Bank/Cycle' button. Note that the emulator has the 'Plug-In' and 'Win' buttons incorrectly reversed.

3 In the fader pack section you should now see that the first 8 channel strips have been assigned exclusively for the purpose of editing your plug-in. Control pages that were laid out vertically in "channel strip" mode are now laid out horizontally across the channels, starting at the bottom and moving upward from left to right.

“Dynamics” and “EQ” sections

If your plug-in explicitly defines page table support for one of these two sections it will automatically appear in the section when inserted, so long as it has focus.

1 Insert your plug-in in Pro Tools.

2 Make sure that it has “focus” (i.e. click on it...).

3 If you’re testing a dynamics plug-in, you should now see its name appear in the “Dynamics” section of the emulator, and should be able to control the plug-in from the encoders in this section. Similarly, if you’re testing an EQ plug-in, you should now see its name appear in the “EQ” section of the emulator.

ProControl Emulator

Although the “hidden” developer page tables menu in Pro Tools is an extremely useful tool for verifying the correct implementation of page tables (and the functions `GetValueString()`, `GetControlNameOfLength()` and `GetControlOrientation()`), the ProControl emulator, which emulates the real ProControl unit in software, can provide valuable interactive feedback. For instance, it is really a necessity to be able to travel through a control’s range to be sure that all control values are handled properly in `GetValueString()`.

The emulator provides this. Contrast this to performing the same thing using the hidden menu in Pro Tools, where each setting of the control's range has to be checked in order to be sure that it is displaying values correctly (i.e., by displaying meaningful results in the 4 characters allocated to it). Performing this using the hidden menu in Pro Tools would be much more time-consuming! The emulator handles this gracefully.

Requirements

To be able to use the ProControl emulator, you'll need to run it on a separate Windows machine than the one that is running Pro Tools. This is so because, like in a real ProControl unit, all communication happens through Ethernet, and the emulator needs to be present in a different physical node over the network. This is both good and bad, since now you'll need yet another machine just to run the emulator but this will more closely resemble how a real ProControl unit will communicate and show up in the Pro Tools Setups/Peripherals window. Note also that you’ll need to use a debug build of Pro Tools for testing your plug-ins using the emulator.

The requirements are as follows:

- A separate PC running Windows XP with an available Ethernet connection adapter in the CPU.
- An Ethernet cable.
- Ethernet DigiDriver installed.
- ProControl .dll is installed in `ProgramFiles\CommonFiles\Digidesign\DAE\Controllers` folder. This is done automatically when you install Pro Tools using the installer.

Setting Up the Emulator

The ProControl emulator can be found on the developer website. Here are the steps necessary to get the emulator working:

1 You'll need to start by installing Pro Tools for Windows off the shipping installer CD, that way you'll make sure you've got the `ProgramFiles\CommonFiles\Digidesign\DAE\Controllers` folder installed.

2 Install the DigiDriver for ProControl Ethernet communication, according to the ProControl User's Manual. For convenience, the steps are mentioned here as well:

To install the Ethernet DigiDriver (Windows XP)

1 Open the Network Connections Control Panel (Start>Settings>ControlPanels>Network Connections).

2 Right-click on the current connection listed and select “Properties”

3 Click Install, select ‘Protocol’, then click Add

4 Click “Have Disk”, then click “Browse” to locate the DigiDriver (\Program Files\Common Files\Digidesign\DAE\Controllers\DigiNet.inf).

5 Select OK. Windows should display "Digidesign Ethernet Support".

6 Select it, click OK to choose, then close the Connection Properties dialog.

7 At the Windows prompt, click Yes to restart the computer.

3 Extract the ProControlEmulator.zip archive to the \Program Files\Common Files\Digidesign\DAE\Controllers\ directory so that the SoftProcontrol.exe file resides alongside DigiNet32.dll.

4 Run the self-extracting installer tcl805.exe (included in this SDK) to install the Tcl/Tk libraries needed. The file is a self-extracting executable. It will install Release 8.0.5 of the Tcl and Tk libraries, the "wish" and "tclsh" programs, and documentation. (Or if you prefer, go to http://dev.scriptics.com/ to download the installer and run it.)

5 Connect the Ethernet cable from the machine with the emulator software to the machine running Pro Tools. If both machines are on the same network this step shouldn’t be necessary.

6 On the emulator machine, run (double-click) the SoftProControl.exe program. The emulator UI should now appear.

7 In the Pro Tools machine, launch Pro Tools.

8 Open or create a session, go to the ‘Setups’ menu, and choose ‘Peripherals’. The Peripherals dialog appears. Select the “Ethernet Controllers” tab, and then check the ‘Enable’ checkbox under Ethernet Port.

9 Next to the blue highlight color, select the unit named "Spike". This is the nickname for the ProControl emulator. Click Ok.

10 If asked to download the current version of the firmware, choose Cancel.

11 The ProControl emulator UI should now reflect your opened session, and you should also have full control over Pro Tools from the emulator. The emulator should behave just like the real hardware unit.

Notice that all rotary knobs have been replaced by sliders. The sliders return to center when released so that they simulate an endless knob. If you need finer resolution try single clicking on the slider scroll bar.

Checking Plug-Ins with the Emulator

Although the emulator can take some time to learn, it is not hard to test plug-ins with it. To instantiate a plug-in on insert one of track one, follow this procedure.

1 Once a session has been created/opened in Pro Tools, on the emulator, press a channel strip's insert switch (labeled "INS/SEND") to insert a plug-in.

2 On the emulator's main section, the insert 1 display will start to blink "No Insert", indicating that it is ready to be assigned a plug-in.

3 Click on the "ASSIGN/ENABLE" button next to the blinking display, it should start to flash red. You can now move the slider at the right of the display to scroll through the available plug-ins list. Again, if you need finer resolution try single clicking at the sides of the scroll bar.

4 After locating with the slider the plug-in you want to insert, click the red flashing assign/enable button. The plug-in should now be inserted and its UI should appear in the Pro Tools screen.

5 To display its parameters (provided the ProControl page tables exist in that plug-in), click on the "INSERT/PARAMS" button at the left of the emulator's main section.

6 You should be now looking at the encoder control values for the first page of the plug-in. Remember that ProControl shows Encoder and Switch values separately, to display the switch values, click the "INFO" button. To edit the Switch values hold and drag the "INFO" button with the mouse towards the display controls in the main section of the emulator (clicking without holding/dragging will only display the switch values momentarily, and you won't be able to change them). To change to the encoders, single click on the "INFO" button again.

7 To change to the next parameters pages (if the plug-in has more than one), click on the green-colored number buttons below the encoder/switch displays.

8 In Pro Tools, you can move the controls of your plug-in, and see the values updated on the emulator. Note that the values displayed on the emulator only change when you release the mouse button. Also, note that in Pro Tools, correctly implemented plug-ins will have blue highlighting as a visual cue, specifying which controls are part of the current page.

Part III: The TDM Architecture

Chapter 7: TDM Hardware

This chapter first describes the HD Accel hardware, followed by descriptions of the older HD Core/Process, and MIX Core/Farm hardware.

PCIe Accel Core and HD Accel Cards

Gershwin III is the codename for Digidesign's PCIe Accel Core and PCIe HD Accel TDM cards. The DSP Core types for these cards are equivalent to those that are found on the HD Accel card, but they differ in number. The following chart compares the allotment of the various DSP core types, which are explained in more detail in the HD Accel Card section below.

	PCI HD Core	PCI HD Accel	PCIe Accel Core and PCIe HD Accel
Presto (256k external RAM)	9	2	3
56321 (no external RAM)	0	3	2
56321 (512k external RAM)	0	4	4

HD Accel Card

Gershwin II is the codename for Digidesign's HD Accel TDM card. As with the previous incarnation of TDM hardware, Gershwin I, this card uses the TDM II architecture, which provides superior performance and efficiency, including 96 and 192 kHz sample rate support. Gershwin II retains the standard PCI card format but replaces seven of the nine Motorola Presto chips from the Gershwin I card with 220 MHz Motorola 56321 DSPs. In addition to more than doubling the clock-rate, these chips provide more than twice the external SRAM and more than ten times the internal SRAM of the previous Presto chips.

TDM II ASIC

TDM II is the core of communication between all Gershwin II system elements. Because TDM II has not changed since Gershwin I, developers of G2 plug-ins need not worry about any underlying card channel or time slot differences. The fact that Gershwin II uses the same TDM II bus architecture and physical TDM connector as its predecessor means that older Gershwin I cards will be compatible on G2 systems.

Gershwin II PCI Card

Like Gershwin I, Gershwin II is a full-length, 66MHz, 64-bit PCI card. Each Gershwin II card includes two Presto and seven Motorola 56321 DSP chips. Each 56321 (henceforth, the '321') runs at 220 MHz, providing about 2.2 times the raw MIPS of the Presto chip, and giving a Gershwin II card a total of 1740 MIPS, or 1.9 times the total MIPS power of a Gershwin I card.

A plug-in instance will reside on one of three possible chip/ram configurations on a G2 card:

- Two 100 MHz Presto DSPs, each with 256k 24-bit words of external SRAM and 15k of internal SRAM. These chips have the same specifications as the Prestos on the Gershwin I.
- Four 220 MHz Motorola 56321 DSPs, each with 512k of external SRAM and 192K of internal RAM.
- Three 220 MHz Motorola 56321 DSPs, each with 192K of internal RAM and no external SRAM.

Each Gershwin card will be equipped with 1 DigiSerial port. This RS-422 based serial port will be used for Sync I/O communication and MachineControl. As with Gershwin I, it is unlikely to be available for general use (you won't be able to hook up a serial MIDI interface).

Memory Configuration

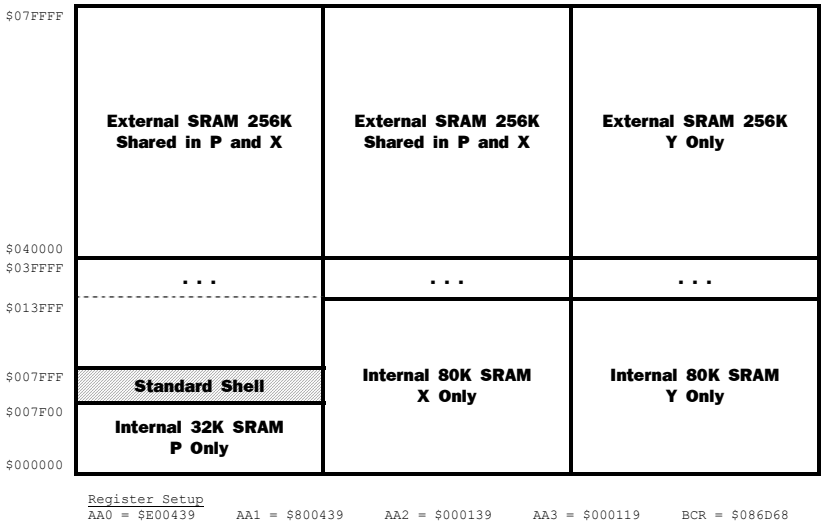
As mentioned, the Gershwin II card has three different possible chip/memory configurations. 321 chips with external SRAM, 321 chips with no external SRAM, and legacy Presto chips, which have the same memory configuration as on Gershwin I.

Internal Memory All seven 321 chips have 32K of usable internal P memory, 80K internal X memory, and 80K internal Y memory.

External Memory Four of the 321s have 512K of external SRAM. Under the default memory map, initialized by DSI, 256K of this memory is shared between P and X. The remaining 256K of memory is available in Y space.

The following simplified memory map illustrates the crucial sections of memory, but does not show aliased memory blocks. A more complete Gershwin II memory map is located at the end of this section.

Simplified Gershwin II Memory Map



The memory map for the 'no SRAM' 321s is the same as the above, excluding the top 'External SRAM' portions.

Standard Shell

The location of the Standard Shell is still in the highest section of internal SRAM, from `$7F00` to `$7FFF`. As always, make sure not to step on this block of memory. Note that under the normal DSP code loading process, a plug-in's DSP code is limited to 32K in size. To load more than this, a workaround will be necessary.

Technical Considerations

Cycle Counts Cycle counts have increased significantly with the new 321 DSPs. At 48kHz, the TDM interrupt has roughly 4400 cycles of processing time. At 96kHz there are 2200 available cycles, and at 192kHz, 1100 cycles. Keep in mind that sample rate pull-up might occur due to SMPTE syncing. This means that at 48kHz your algorithm must be able to accommodate a sample rate of 50kHz. Likewise, at 96kHz, as high as 100kHz. Or at 192kHz, up to 200kHz. The TDM interrupt routine must complete its processing within these worst-case scenarios.

TDM I/O As with Gershwin I hardware, there is a fixed interval at the beginning of the TDM interrupt within which all I/O communication with the TDM ASIC must be completed. We encourage that you implement the double-buffering scheme so that the ASIC setup time requirements are always met.

External SRAM Wait States 4 wait states are necessary for accessing the external SRAM. **Note that this is contrary to what was previously published regarding the number of wait states**. Note also that due to the single bus connecting the chip to external memory, a parallel move instruction (which would normally take 1 cycle in internal memory) will require 10 cycles if accessing external memory! The BCR register is initialized with wait state settings prior to the loading of a plug-in's DSP code.

Interrupt Vectors The interrupt vector table of the 321 is slightly different than Presto. However, this shouldn't affect any DSP code that uses the `StdDSP321Defs.asm` include.

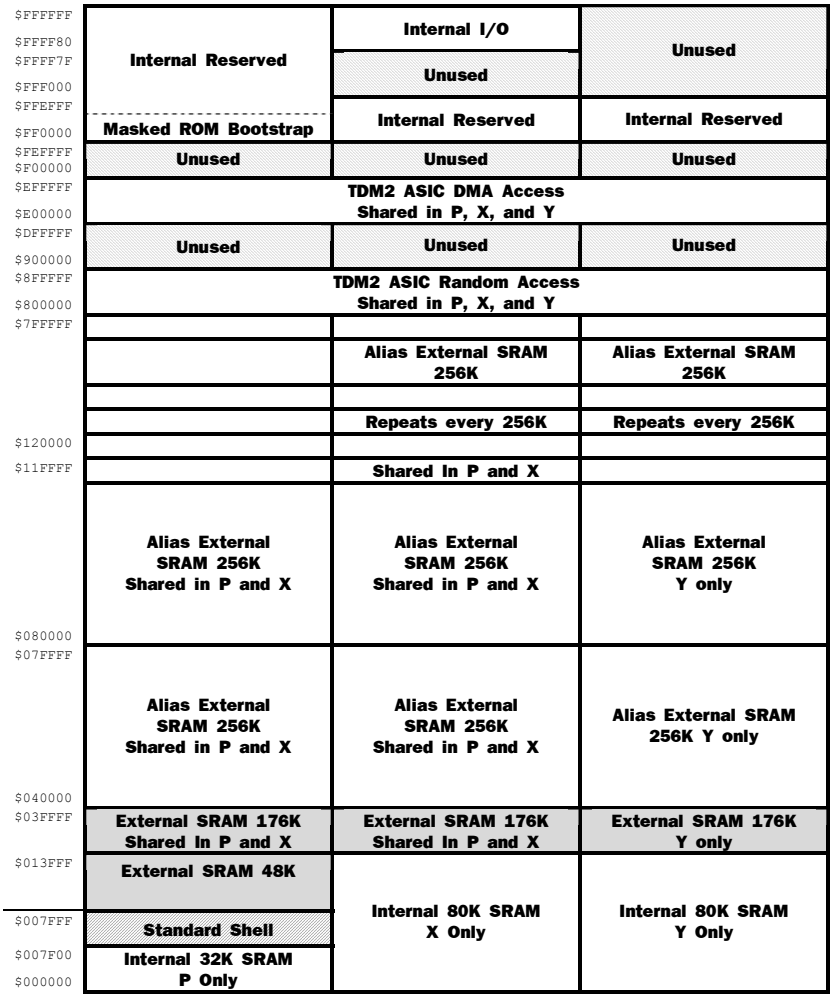
The DMA controller for the 321 does not have the same bug as is documented for the Presto. Note that this is contrary to what was previously published regarding the 321s DMA controller. One difference in DMA between Presto and 321 is that on the 321 DMA cannot address memory below the 12k boundary. Another difference is that the 321 has 1024 word DMA memory pages rather than 256. Check out page 4-29 of the 321 manual for more info on DMA on the 321.

The Host Port The 321 uses the HI08 rather than the HI32 host port of the Presto. See the Motorola documentation for more details. Access to the host port has been abstracted in G2. This should be transparent to developers who use the normal SendLoLong(), GetLoLong() type of accessor methods to talk to the DSP. Developers who use custom code to communicate directly with the host port (CDSP::fHostPort) should update their code to use the new abstraction layer. Please contact Developer Services if you require assistance.

Memory Map

The following table shows the complete Gershwin II memory map as initialized by DSI. By manipulating the AAx registers it is possible to change the 321's memory mapping from this default. The SRAM is seperated into two equal banks. When the requirements of AAR2 are met, one bank of the external SRAM is enabled (128K). AAR3 enables the other bank. Furthermore, the 563xx architecture specifies that AAR3 has priority over AAR2.

Complete Gershwin II Memory Map



Register Setup
AA0 = $E00439 AA1 = $800439 AA2 = $000139 AA3 = $000119 BCR = $086D68

HD Core/Process System

Jazz is the codename for Digidesign's fourth generation Pro Tools TDM system. This system is designed around the TDM II audio bus architecture which provides increased performance and efficiency, including 96 kHz and 192 kHz sample rate support. The Jazz system retains the standard PCI card format, as the previous MIX hardware, but replaces the six Motorola Onyx chips per card with nine 56k-compatible DSPs, codenamed Presto. Additionally, the system includes new I/O interfaces and a standard I/O cable for connecting to those interfaces.

TDM II ASIC

The TDM II ASIC is the basis of our next generation TDM architecture. TDM II, which offers significantly more timeslots than TDM I, is the core of communication between all Jazz system elements.

The bus architecture for TDM II has changed considerably. The original TDM system had a single bus to which all DSP chips were connected. With TDM II, DSP chips are daisy-chained together. Inter-chip communication passes down the chain in either direction. This allows for "spatial reuse", achieving even further timeslot efficiency. A single timeslot can be used at the same time for communication between multiple neighboring DSPs. In addition, each DSP has a virtually unlimited number of card channels, allowing more I/O paths to and from the chip. Though this new TDM architecture is significantly more sophisticated, this complexity is hidden and mostly transparent from a development standpoint.

Gershwin PCI Card

Gershwin is the codename of the PCI card. It is a full-length, 33MHz, 64-bit PCI card. Similar to MIX, two versions of the card are available, "Core" and "Farm".

Each Gershwin card includes 9 Presto DSP chips. Each Presto chip runs at 100 MHz providing about 1.2 times the raw MIPS of the Onyx chip, giving a Gershwin card roughly 1.9 times the MIPS power of a MIX card.

Unlike MIX, on a Gershwin card, all the Presto chip's resources are identical. Each provides 256K 24-bit words of SRAM. Fortunately, this will help simplify the plug-in "shuffling" procedure with respect to MIX. Unfortunately, the absence of DRAM will introduce a challenge for long-delay plug-ins. Please contact Developer Services if your product requires DRAM support.

Gershwin cards also include a TDM connector that is mechanically different from the old ribbon system. Because of the TDM II bus architecture and new connector, older MIX and D24 cards are not compatible with Gershwin cards.

Each Gershwin card is equipped with 2 DigiSerial ports. These RS-422 based serial ports will be used for USD II communication and MachineControl. They run at a significantly higher baud rate than the ports on the MIX cards. Also, like MIX, they are unlikely to be available for general use (you won't be able to hook up a serial MIDI interface).

Memory Configuration

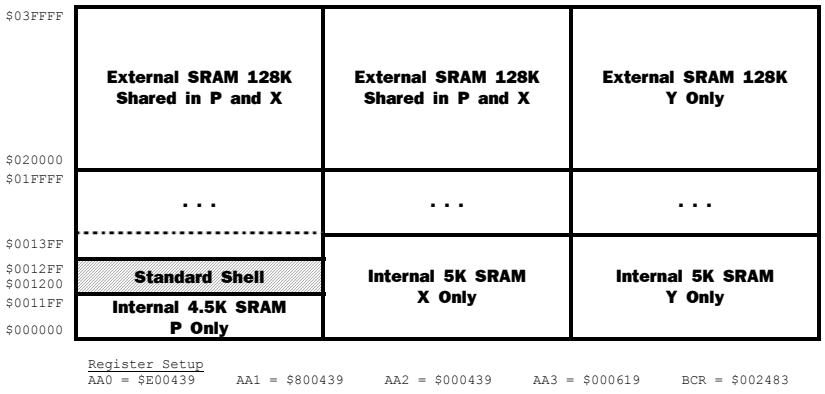
The memory map for all nine DSPs is identical with each having 256K words of external SRAM. As implied, none of Gershwin's DSPs have attached DRAM.

Internal Memory Each DSP has 4.5K of usable internal P memory, 5K internal X memory, and 5K internal Y memory.

External Memory Accessible by each DSP is 256K of external SRAM. Under the default memory map, initialized by DSI, 128K of this memory is shared between P and X. The remaining 128K of memory is available in Y space.

The following memory map illustrates the crucial sections of memory, and does not show the aliased memory blocks. A more complete Gershwin memory map is found later in this document.

Simplified Gershwin Memory Map



Standard Shell

The location of the Standard Shell has moved. It is now at the highest section of internal SRAM, from `$1200` to `$12FF`. As always, make sure not to step on this block of memory. Note that under the normal DSP code loading process, a plug-in's DSP code is limited to 4.5K in size. To load more than this, a workaround will be necessary.

Technical Considerations

Cycle Count Guidelines The HD system supports a new pull-up of 4.1667% (25/24) in addition to a 0.1% pull-up. When using both 4.1667% and 0.1% pull-ups at the same time, the maximum system sample rate becomes 50.05kHz, assuming a base rate 48kHz, or 100.1kHz and 200.2kHz at 96kHz and 192kHz base sample rates. Allowing for an extra 4% of headroom to account to host port communications, interrupt jitter, etc., the following maximum cycle counts for the Gershwin card were developed.

Sample Rate	Max DSP cycles
48kHz	1920 cycles
96kHz	960 cycles
192kHz	480 cycles

TDM I/O As with MIX hardware, there is a fixed interval at the beginning of the TDM interrupt within which all I/O communication with the TDM ASIC must be completed. Unfortunately, at the new higher sample rates, this window has grown even smaller. We encourage that you implement the double-buffering scheme so that the ASIC setup time requirements are always met.

External SRAM Wait States There is one bus wait state necessary for accessing the external SRAM. The BCR register is initialized with wait state settings prior to the loading of a plug-in's DSP code.

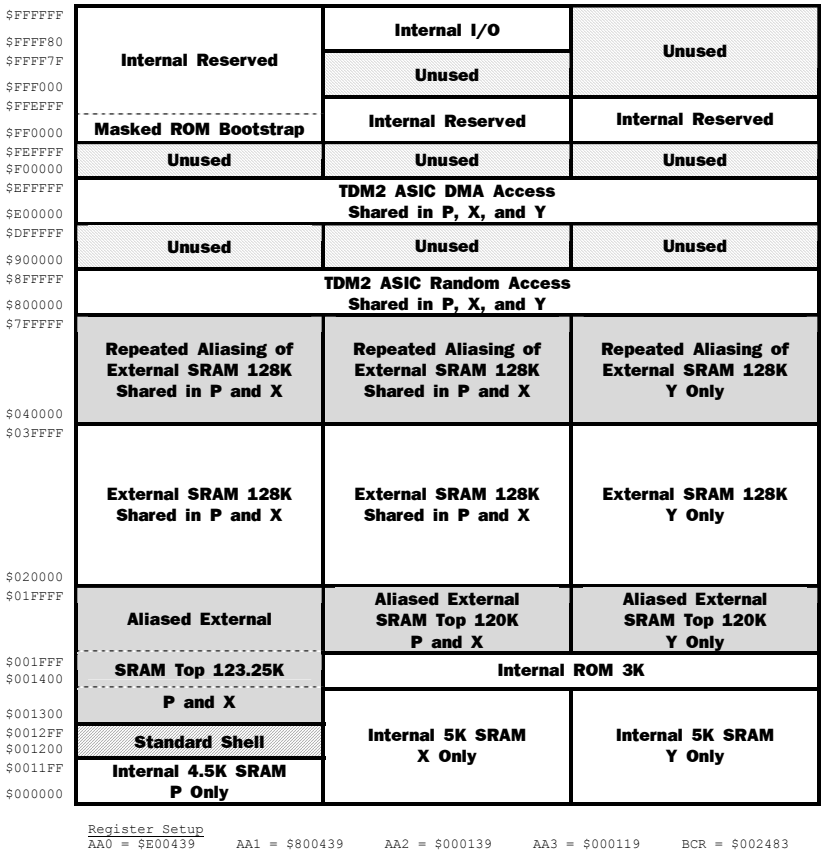
Interrupt Vectors The interrupt vector table of the Presto is slightly different than Onyx. Of note is the new location of the Host Command interrupt vector. However, this shouldn't affect any DSP code since it's encapsulated in the `StdPrestoDefs.asm` include file and reflected in the `HostCmdInt`*X* defines.

Presto's DMA Controller is Buggy! Please contact Digidesign Developer Services for details.

Memory Map

The following table shows the complete Gershwin memory map as initialized by DSI. By manipulating the AAx registers is possible to change the Presto's memory mapping from this default. The SRAM is seperated into two equal banks. When the requirements of AAR2 are met, one bank of the external SRAM is enabled (128K). AAR3 enables the other bank. Furthermore, the 563xx architecture specifies that AAR3 has priority over AAR2.

Complete Gershwin Memory Map



Register Setup
AA0 = $E00439 AA1 = $800439 AA2 = $000139 AA3 = $000119 BCR = $002483

MIX Core/Farm System

The MIX hardware comes in two flavors: MIX Core and MIX Farm. The MIX Core can replace the d24 card of a Pro Toos|24 system while still providing additional DSP for signal processing, and the MIX Farm is available for sheer Plug-in DSP power. For the purposes of this SDK, both of these cards will be referred to as the MIX Farm (since there is no difference from the standpoint of plug-in development).

Each MIX card contains six Motorola 56301 (Onyx) DSP chips. Each DSP communicates to the TDM bus through a Digidesign TDM ASIC (Application Specific Integrated Circuit). Each DSP runs at 80 MHz.

The card achieves 480 MIPS (80 MIPS per DSP) compared to 132 MIPS (33 MIPS per DSP) on the Merle PCI DSP Farm and 80 MIPS (20 MIPS per DSP) on the Nubus DSP Farm. Two of the DSP chips have 2 Mbytes of DRAM to facilitate time-domain processing not possible on previous DSP Farms.

Three of the DSP chips have 128K words of SRAM. One of the DSPs has no external memory. The design choices for memory were made to create a card design that maximized the number of DSP chips, provided plenty of audio delay, provided backward compatibility with existing plug-in designs, and provided a competitive cost point.

Internal Memory

All the DSPs have 2K internal P memory, 3K internal X memory, and 3K internal Y memory. Compared to the 56002's internal memory, this is 4 times P memory, 12 times X memory, and 12 times Y memory. This increased internal memory reduces or eliminates the need for external memory for many plug-ins such as EQ, Dynamics, and Mixing. It also reduces or eliminates DSP overhead incurred by wait states required to access external memory.

External Memory

Three of the DSP's have 128K words of SRAM. Like the Merle PCI DSP Farm, the memory is divided into two 64K word blocks. The Y space is not shared with any other space (except for the small top 3K amount, which is shared with P space). The X and P spaces are shared memory. Unlike the Merle PCI DSP Farm, the TDM ASIC address space does not conflict with Y memory space. The 56301 has a 24-bit address space so now the TDM ASIC is mapped to a memory location much higher than occupied by the SRAM. Also unlike the Merle PCI DSP Farm, the Standard Shell (i.e., the SA Driver) code is not located within the lower 32K of P memory. It is now located just below the 64K boundary of X/P memory. Please refer to the MIX Farm Memory Maps for more information.

DRAM

Two of the DSPs have 2 Mbytes of DRAM. This allows up to 14.5 seconds of audio delay at 48kHz sample rate per DSP chip. Direct byte-wide accesses can be performed to read or write words of data or the 56301 DMA controller can be programmed to access this memory.

The 56301 has a feature that allows a low-cost design using a 2Meg X 8-bit chip to be connected with no 'glue.' The 56301 DRAM controller has a feature that automatically packs and unpacks 24-bit words in the 8-bit memory. This feature is available by using the DMA controller to access the DRAM. Therefore, one strategy for writing DSP code to use DRAM is to program the DMA controller to asynchronously transfer data between the DRAM and internal SRAM.

Using DMA for TDM I/O

The 56301 DMA controller allows for more efficient access to TDM I/O. The existing Merle PCI DSP Farm requires the TDM chip to be directly accessed. Because DSP wait states are necessary to access it, a lot of DSP horsepower is consumed by plug-ins attempting to process a lot of channels of audio. The 56301 DMA controller can be programmed to asynchronously read all 64 channels of TDM audio input and place them in internal DSP memory for fast access by the DSP code. And, the DMA controller can simultaneously write all 64 channels of TDM output from internal DSP memory.

Plug-In Development Strategies

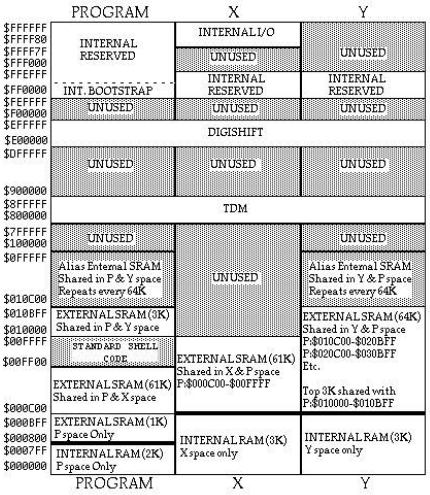
Different types of plug-ins have different memory needs. The MIX Farm provides a balance of memory designs to facilitate different types of Plug-ins. If a Plug-in needs small amounts of memory, the design goal should be to use no external memory. This allows a plug-in to run on any of the 6 DSP's. Examples are EQ and Dynamics. If the Plug-in needs large amounts of external memory and needs to access many locations per sample frame, it should be designed to operate as many instantiations as possible on a DSP chip with SRAM. Examples are complex reverbs and FFT processors. If a plug-in needs large to huge amounts of external memory, but needs to access a small number of words per sample frame, it should be designed to operate as many instantiations as possible on a DSP chip with DRAM. Examples are delay lines, chorus, flanging, and simple reverbs.

Memory Maps

On a MIX card, the six DSPs are labeled as follows:

DSP A External SRAM, labeled A in Allocator
DSP B No external RAM, labeled B in Allocator
DSP C & F External DRAM, labeled C in Allocator
DSP D External SRAM, labeled A in Allocator
DSP E External SRAM, labeled A in Allocator

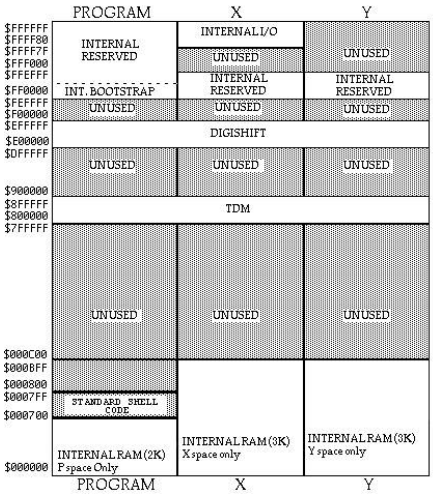
DSP "A" Memory Map



DSP A: External Devices
TDM: AA0 = $800439
DigiShift: AA1 = $E00439
SRAM: AA2 = $000439 (SRAM enable), AA3 = $000819 (SRAM address line 16)
Additional Hardware Setup
Memory Switch in the OMR Register is set (Bit 7=1)
Address Priority Disable in the OMR Register is set (Bit 14=1)
BCR Register (@ 80MHz) = $002444 (DigiShift I/O), $002504 (DigiShift Configuration)
DCR Register (@ 80MHz) = $000000

9/16/98 Rev. 4/11/2000

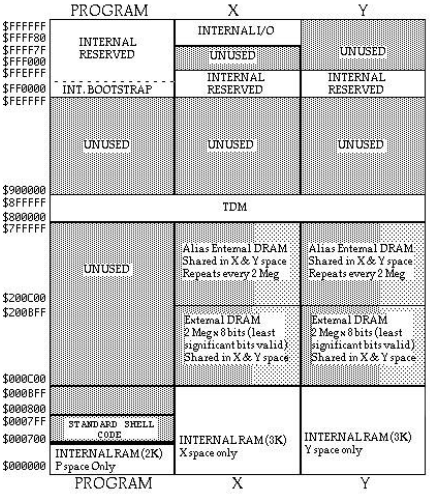
DSP "B" Memory Map



DSP B: External Devices
TDM: AA0 = $800439
DigiShift: AA1 = $E00439
Unused: AA2 = $000000 AA3 = $000000
Additional Hardware Setup
Memory Switch in the OMR Register is set (Bit 7=1)
Address Priority Disable in the OMR Register is set (Bit 14=1)
BCR Register (@ 80MHz) = $000044 (DigiShift I/O), $000104 (DigiShift Configuration)
DCR Register (@ 80MHz) = $000000

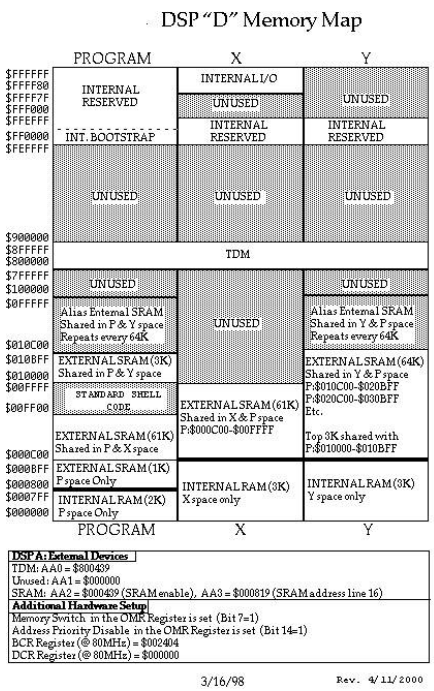
11/4/97 Rev. 4/11/2000

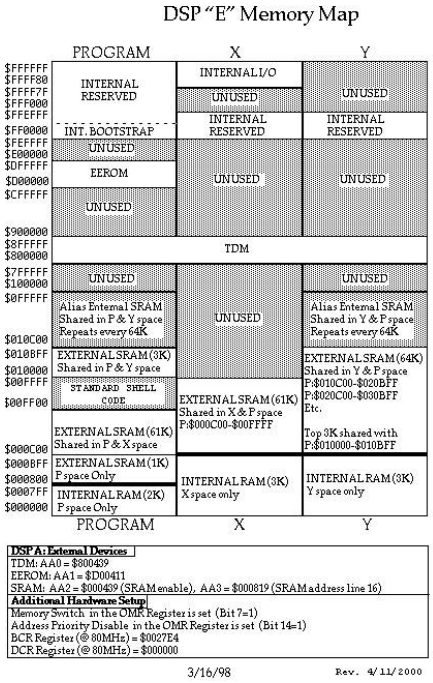
DSP "C" & "F" Memory Map



DSP C & F: External Devices
TDM: AA0 = $800439
DRAM: AA1 = $0001B2
Unused: AA2 = $000000, AA3 = $000000
Additional Hardware Setup
Memory Switch in the OMR Register is set (Bit 7=1)
Address Priority Disable in the OMR Register is set (Bit 14=1)
BCR Register (@ 80MHz) = $000004
DCR Register (@ 80MHz) = 89390A

4/10/98 Rev. 4/11/2000





SRAM DSP Maps (A, D, & E)

This section can be read to gain a clearer understanding of the SRAM layout. These DSPs have two separate 64K blocks of external SRAM available which are divided among the X:, Y: and P: memory spaces (except for a small 2K amount from $0 to $0007FF in the X:/P: space which cannot be accessed by the DSP). External memory in some cases is "shared" and/or "aliased" among the spaces, as explained shortly. Note also that 1 wait state is required for any external memory access to SRAM (and 4 wait states are required to access the TDM chip).

Much of the X:/P: memory is shared. This means that the same logical address in different memory spaces actually references the same physical location in external SRAM. For example with X:/P: memory, X:$000C00 and P:$000C00 reference the same physical SRAM location. In SRAM chips, X:/P: memory is shared in the range of $000C00 to $00FFFF. X:/P: memory is not shared in the 1K program memory from P:$000800 to P:$000BFF and as noted above, the physical X:/P: external memory from $0 to $0007FF cannot be accessed, since this range accesses the internal X:/P: memory.

The external SRAM in the X: space cannot be accessed above X:$00FFFF because of a hardware restriction that prevents aliasing in this memory space. Aliasing is where different logical addresses in the same memory space reference the same physical location in SRAM. (For example, Y:$000C00 and Y:$010C00 reference the same physical SRAM location.) Note that the physical external Y: memory located in the internal Y: memory range (Y:$0 through Y:$000BFF) is aliased to the range Y:$010000 to Y:$010BFF (and also shared with P memory). Of special importance is the contiguous aliasing of the Y: memory from Y:$010000 to Y:$0FFFFF (which is also shared with P space in the range of P:$010000 - P:$0FFFFF).

Aliasing of the external memory can be used to good advantage in creating large circular buffers starting at Y:$010000 and extending all the way up to Y:$01FFFF if needed (i.e., up to a 64K circular buffer). This is made possible due to the placement of the Standard Shell, and keeps the fast internal memory free for other uses.

NOTE: Of the revisions of the Onyx chip (Rev A, Rev B), Digidesign has never shipped any Rev A chips. Rev B chips actually support two primary internal memory configurations (which is setup in the Booter.asm code); therefore, some Motorola manuals may show a memory configuration different than provided here. Specifically, the 8k total of internal memory may alternately be divided up as: 4k of P memory and 2k each of X and Y memory—a configuration that is not supported at this time.

Location of the Standard Shell

For the SRAM DSPs, the DigiSystemINIT (a.k.a. DSI) places the Standard Shell at `P:$00FF00` to `P:$00FFFF` (256 words are reserved for it), which is also shared with the top of X-memory. Therefore, be careful not to "step" on the Standard Shell in this range on an SRAM DSP. Also note for the other non-SRAM DSPs, the Standard shell is located in the highest portion of P memory, that is: `P:$000700` to `P:$0007FF`.

Accessing DRAM

Direct-Access Method

The DRAM can be accessed one byte at a time, in sequential addresses starting at `X:` or `Y: $C00`. The addresses are "aliased" every 2 MB (e.g., `$200000-$3fffff` was used for the Mod Delay plug-in). When accessing a byte for each read, it is required to "pack" (i.e., concatenate) the bytes to form a 24-bit word. Here is how to use the "insert" and "extract" operations for this:

For writing a 24-bit word in `a1` to the DRAM:

```
move     a1,a0             ;put the input in a0
extractu #kInsertHi8,a,b   ;put the hi 8 bits of a0 into b0
move     b0,y:(r4)-        ;store the hi 8 bits into the DRAM
extractu #kInsertMid8,a,b  ;get the middle 8 bits in b0
move     b0,y:(r4)-        ;store the middle 8 bits into DRAM
extractu #kInsertLow8,a,b  ;get the low 8 bits into b0
move     a0,a              ;restore the value in a1
move     b0,y:(r4)-        ;store the low 8 bits into DRAM
```

where the constants are:

```
kInsertLow8  equ  $008000
kInsertMid8  equ  $008008
kInsertHi8   equ  $008010
```

and r4 points to the DRAM space ($200000). Similarly for reading the DRAM, the following can be used:

```
move     y:(r4)-,x0        ; get hi 8-bit value
insert   #kInsertHi8,x0,b  ; put in to b0
move     y:(r4)-,x0        ; get middle 8-bits
insert   #kInsertMid8,x0,b ; insert into b0
move     y:(r4),x0         ; get low 8-bits
insert   #kInsertLow8,x0,b ; insert into b0
move     b0,x0             ; store result in x0
```

DRAM Wait States

Note that wait states and DRAM control registers are already set by DSI, so the plug-in should not need to set them. For in-page access to DRAM, there are 3 wait states, and out-of-page access is 11 wait states. Using the default page size of 512, this would assume that the first access is out-of-page and subsequent accesses in the same area (within 512 bytes) are in-page. The refresh rate is every 15.24 ms. Therefore, assume that you are starting with an out-of-page access for each new sample period.

DMA Controller DRAM Access Method

DMA channels can also be set up for reading the DRAM, and they can automatically "pack" the 8-bit values together. Please refer to the Onyx books for a complete discussion on DMA channels (specifically, chapter 8 of the 56300 book). Though here is a simple code snippet for setting up a DMA channel for reading or writing the DRAM:

```
movep #>0,x:M_DCO0               ; set counter to 1 word for DRAM reads
movep #>dramAddress,x:M_DSR0     ; get DRAM address as source
movep #>internalBuf,x:M_DDR0     ; get internal values buffer as
                                 ; destination
movep #kAccessDRAMWord,x:M_DCR0  ; trigger the DMA channel read
```

where:

```
kAccessDRAMWord    equ   $9E0245    ; DMA Control register for DRAM accesses
```

This sets the source and destination for the DMA access and then triggers the transfer by setting the DCR register. The bytes get "packed" automatically, so this will transfer 3 bytes into a single DSP word.

Worst-Case Sample Rate Design

To be within specification, plug-ins must be designed for a worst-case sample rate of 50 KHz (not 48 KHz). This is the scenario when a 48 KHz sample rate is running in a system utilizing SMPTE with "pull up," which pushes the TDM bus up to a rate of 50 KHz.

Following are the cycle counts available in the TDM interrupt for processing on the different Farm cards:

Nubus Farm	40 MHz / 50 kHz = 800 cycles
Merle PCI Farm	66 MHz / 50 kHz = 1320 cycles
MIX PCI Core/Farm	80 MHz / 50 kHz = 1600 cycles

The TDM interrupt loop must not exceed the number of clock cycles given above for the specified DSP! Due to pipelining and other optimizations, the Onyx chip performs much more efficiently in the clock cycles available.

MIX Summary

The importance of a well thought out memory management scheme for your plug-in can not be understated, since with a shared memory system such as this, it is very easy to overwrite memory that should not be overwritten (e.g., program memory!). By taking the time to fully understand how the X:, Y:, and P: spaces share the external memory, you can quite possibly save countless hours of 56K code debugging. Therefore, forethought and planning is time well spent.

The MIX Farm is a very powerful design, but more complex than the previous DSP Farm. It offers many more MIPS and much more memory. A plug-in's use of DRAM differs greatly from any previous plug-in

design. A plug-in developer should carefully understand this hardware to develop plug-ins that will be in great demand by the professional and project studio markets.

Chapter 8: Writing TDM DSP Code

There are two methodologies for developing DSP code to run in the TDM system. The older approach involves programming the DSP's entire operations, from the interrupt vector table to all the interrupt handling routines, which provide host-to-dsp communication. Even though many of these routines are standardized, since they're necessary for communicating with the TDM system and Plug-In Library, this approach is tedious. In addition, its DSP inefficient, since the plug-in must take over an entire DSP chip, even if only one out of 50 possible instances is running. The result: the end-user is only able to run a handful of different manufacturer's plug-ins at the same time.

To address these problems, MultiShell was developed. MultiShell accomplishes the following:
1 Reduces the DSP coding workload to its bare minimum. Using MultiShell, a developer need only code the DSP algorithm, and the communication methodology with the host code.
2 Abstracts, simplifies, and standardizes as much as possible the context in which the DSP code runs.
3 Allows multiple plug-ins from different developers to share a single DSP chip.
4 Potentially allows easy portability to Digidesign's future 56K-based hardware systems.

This chapter first discusses the MultiShell system. The second half describes the original approach for DSP coding. Within MultiShell, the audio streams that the algorithm processes are "spoon-fed" to the plug-in in buffers; in contrast, the old approach requires much more of the developer in terms of communicating audio within the TDM system. Therefore, the details of TDM are held off until later; but, if you're curious, skip ahead to **Non-Mush TDM Development**.

Lastly, this chapter assumes you are intimately experienced programming the Motorola 56K DSP architecture.

MultiShell II

MultiShell I was a DSP sharing technology used previously (pre-Pro Tools 5.1) and exclusively by some core Digidesign plug-ins. This mechanism was then exhaustively revised and polished for third party developer consumption, yielding MultiShell II. The MultiShell II (a.k.a. MuSh) is a technology that lets multiple instances of *any* MultiShell plug-in share a common DSP. In addition, MuSh presents a basic abstraction layer/shell to the plug-in's DSP code. This relieves the developer of DSP resource management tasks, and streamlines the overall DSP code development process. However, using a DSP in such an unconventional manner places some constraints on a MuSh plug-in:

- The DSP code must be written to receive pointers to its X, Y, & L memory at run-time rather than using hard-coded constants.
- Absolute addressing must be avoided.
- Since MuSh itself uses the DMA controller, this pre-empts the plug-in code from any DMA usage.
- A MuSh plug-in is also prevented from doing anything that can't be saved and restored within a single sample interrupt. This includes background or block processing.
- Furthermore, since the host plug-in code can only communicate with the "shell" of MultiShell, there is no allowance for direct communication with the DSP code. All communication must be framed in terms of "Getting" and "Setting" memory that the DSP process owns.
- `MuSh plug-ins cannot use Hardware Copy Protection (HCP).`
- Lastly, there are a handful of 56K instructions that are forbidden or restricted in the MuSh system.

MuSh's Unsupported/Discouraged 56K Opcodes

The following list of opcodes should not be used in a MuSh plug-in's DSP code.

PLOCK	MOVEM	DEBUGcc
PUNLOCK	MOVEP	DEBUG
PLOCKR	MOVEC	TRAPcc
PUNLOCKR	DO FOREVER	TRAP
PFREE	RESET	RTI
PFLUSH	STOP	
PFLUSHUN	WAIT	

Relocation of loop and Jump addresses

Instructions like DO and all the JUMP-type instructions (Jcc, JMP, JCLR, JSET, JScc, JSR, JSCLR, and JSSET) are supported through a relocation scheme in MultiShell. This relocation scheme translates the intended DO or JUMP address to its actual physical equivalent in the MultiShell environment. Although they are supported, it is recommended that the developer use all their relative equivalents instead, if supported by the chip, e.g. use DOR instead of DO, and BRANCH-type (Bcc, BRA, BRCLR, BRSET, BScc, BSR, BSCLR and BSSET) instructions instead of JUMP-type. Unfortunately, none of these are available in Merle hardware. At some point in the future, we will stop supporting Merle hardware, and we may remove support of absolute addressed branch instructions. Also, the address relocator is not 100% guaranteed to work in all cases. The main case where this can occur is if the relocator misinterprets immediate data as a valid instruction. While much effort has been made to minimize the occurrence, it could still happen in some rare situations.

The DSP Code Context

Now that the caveats of MuSh are out of the way, let's look at how to use it. In a TDM system, MuSh or non-MuSh, every DSP Process is allowed to do processing at every sample period. With MuSh, a single DSP code fragment is used for all Processes and audio channels running under a specific Type. Therefore, at every sample period, the algorithm embodied in this code segment must iterate over all incoming audio streams, using the stream's corresponding set of parameters and state. To locate the input and output buffers, the number of streams, and the memory locations of any state/parameter blocks, a pointer is passed in by MuSh which points to a table of information. Register R0 is used. The default configuration of the table is as follows:

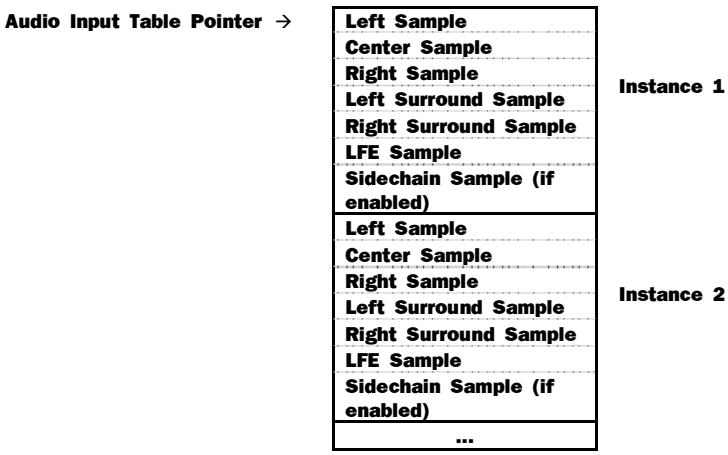
R0 →	**Audio Output Table Pointer**
	Audio Input Table Pointer
	Total Number of Output Streams
	X/Y/L Memory Block 1 Pointer
	X/Y/L Memory Block 2 Pointer
	X/Y/L Memory Block 3 Pointer
	X/Y/L Memory Block 4 Pointer

On the host, this table's layout is defined at the Process level in the method CMultiShellProcess::MuShInitPerSpecies(). By overriding this method, it's possible to restructure, and alter the values that are passed in. The default implementation is shown here.

```
void CMultiShellProcess::MuShInitPerSpecies (const MuSh::SInstanceInfo* iArgs)
{
     long aArgMem [MuSh::cNumUserArgs] =
     {
          iArgs->mOutAudioAddr,
          iArgs->mInAudioAddr,
          iArgs->mTotInst * GetNumAudioOutputsFun (),
          iArgs->mBaseAddr [cArgLoc1],
          iArgs->mBaseAddr [cArgLoc2],
```

```
            iArgs->mBaseAddr [cArgLoc3],
            iArgs->mBaseAddr [cArgLoc4],
            0
        };

        SetArgMem (aArgMem);
}
```

The input table entry points to the memory location containing the blocks of input samples. Every instance's block is contiguous with the next. For example, for a 5.1 surround Type, the table would have the following table layout.



If sidechain inputs have been enabled for the Type then its sample slot will simply be inserted last into the array of samples for the instance. If the sidechain hasn't yet been connected in the session by the user, the sidechain sample will contain zero.

The output table has the same form; with the exception of sidechains, which do not stream out of a plug-in.

"Total number of output streams" is a value equal to the number of instances times the number of output audio channels per instance, i.e. the total number of output samples that the DSP code must generate during the sample interrupt.

The memory block pointers are pointers to either X, Y, or L memory that the MuSh system has allocated for the plug-in to use. In the host-side code, the memory requirements of the DSP code can be registered with the MuSh system. This is described in the next section.

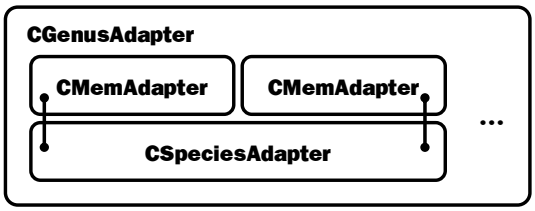
Register Usage

If your plug-in has changed an address modifier register, you must reset it back to linear addressing mode at the end of the TDM interrupt. If you wish to protect your plug-in against a modifier register that has not been reset, you can reset it manually at the beginning of the interrupt.

The MuSh Classes

Several MuSh classes exist that are used to build up a description of your plug-in's resource requirements on the varied TDM hardware platforms. These classes are used at the Type level and are exercised in the method `MuShGetInfo()`.

Using the Effect Layer, MuSh definition is done in the `CreateEffectTypes()` *method at the Group Level.*



The above graph represents the relationship between the three MuSh classes. The Genus Adapter represents an abstraction of the DSP code. Any number of Mem Adapters, which specify the logical memory requirements of the DSP code, can be added to the Genus. In turn, each Species, which represents the physical resource requirements of the DSP on a specific hardware platform must be added. This species must have a physical memory description that corresponds with the set of logical Mem Adapters.

CMemAdapter

A Mem Adapter represents one single piece of X, Y, or L memory that the DSP code will utilize for storing its state and parameter. This single piece of memory is used for all instances of the plug-in. MultiShell insures that all instance blocks will be contiguous in memory, and will not become fragmented over time as plug-ins are inserted and removed. Its public members are initialized as follows:

- `CMemAdapter::mArgLoc`

This is a unique ID associated with this `CMemAdapter` object. For convenience you can use `cArgLoc0`, `cArgLoc1`, `cArgLoc2`, `cArgLoc3`. Essentially, `mArgLoc` describes its placement in the pointer table that is passed into the MuSh DSP code during runtime. (This will be described in better detail later.)

- `CMemAdapter::mLenPerInst`

This is the factor by which the chunk of memory should grow per instance.

- `CMemAdapter::mLenPerSpecies`

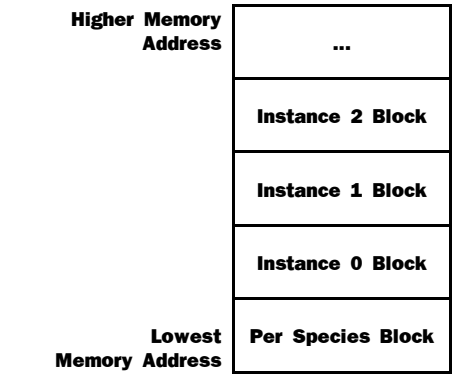
Defines a global chunk of memory for all instances that is allocated at the lowest memory address, before the instance's memory blocks. When `SetMem()`ing and `GetMem()`ing this block, be sure to specify an offset (`iOffs`) of zero and an `iIsInstRel` argument of false. Otherwise, MultiShell would appropriately offset to the "PerInst" block.

- `CMemAdapter::mMemSpace`

Set equal to eX, eY, or eL to specify which type of memory this block accesses.

- `CMemAdapter::mModulusLog2`

This is the power of two that you need for Modulo addressing. This would be equal to 4 for modulo addressing of 16 (i.e. 2^4).



After a Mem Adapter has been initialized, it is "pushed back" into a `CGenusAdapter` which is described later.

CSpeciesAdapter

A Species adapter describes the particular chip you can instantiate your plug-in on, for a particular type of hardware (Merle = DSPFarm, Satchmo = MIXFarm), but even more so, it represents a specific type of RAM you will need.

■ `CspeciesAdapter::mChipSet`
This is a mask-based indicator of the type of chip(s) your plug-in can run on. For example, returning `CChipConstants::MixAll()` here would mean your plug-in can run on any given chip in MIX hardware. Returning `CChipConstants::MixSRAM()` means your plug-in can only run on SRAM chips. See ChipConstants.h for the class definition and usage.

■ `CspeciesAdapter::mCycles`
Here, a utility is provided that calculates for you the actual number of cycles your plug-in uses on any given hardware. You use it as:

`CCycleCount(a, b, c)`, where:

a = cycle count at the per-process "DO" instruction
b = cycle count at the last instruction of the per-process "DO" loop
c = cycle count at the last instruction

You can read cycle count in your .lst file generated when compiling the .asm file, assuming you enable it by using the 'cc' option in it. See `CMultiShellProcessType.h/.cpp` for a full description.

■ `CSpeciesAdapter::mCycles.mPerInst`
Number of cycles (mCycles) multiplied by the number of outputs this ProcessType has.

■ `CSpeciesAdapter::mRealInstRAMTypeList.mX.push_back(ERAMType)`
This represents the exact type of memory you want for your plug-in. Internal (`eIRAM`), external SRAM (`eSRAM`), or DRAM (`eDRAM`). You must "push back" a type for every Mem Adapter you plan to push back into the Genus Adapter, and in the corresponding order.

CGenusAdapter

A Genus adapter represents a piece of DSP code object, independent of the kind of physical RAM it will run on; instead, it specifies the logical types of memory it uses, i.e., X, Y, modulo, etc, by the Mem Adapters that are added to it.

`CGenusAdapter::CGenusAdapter (int iResourceID, const char* iName)`
In the constructor of this object, you specify the actual DSP code resource number, corresponding to a particular compiled `.asm` file you use for this ProcessType and type of RAM (assuming you write different versions of DSP code that run on different types of chips).

`CGenusAdapter::mRealSpeciesList.mX.push_back(CSpeciesAdapter &)`
Here all Species objects are pushed back for this Genus.

`CGenusAdapter::mRealMemList.mX.push_back(CMemAdapter &)`
Here the Mem Adapters, the memory blocks, that all the species will use are pushed back.

The order in which you "push_back" the Mem Adapter objects in the Genus must be the same as the order you "push_back" the RMemType in your Species.

MuSh Plug-In Library Methods

Initializing MuSh Processes

At the Process level, the method `MuShInit()` is invoked by DSPManager anytime the state/parameter memory of your MultiShell Process needs to be fully initialized. Essentially, this method is invoked every time a new MuSh plug-in is inserted in a session, since a potential reshuffle may have occurred. The parameter data should not be reset but, rather, updated to the currently set values.

Talking to Your DSP

Following is a descriptions of the available Process level MuSh methods used to do memory transfers between the host and DSP.

```
void SetMem
       (int iArgLoc, long iOffs, SInt32* iData, long iLen, bool iIsInstRel=true)
```

This method transfers data from the host to the DSP. `iArgLoc` specifies the destination memory block within the Process, which was defined previously by a Mem Adapter, e.g. `cArgLoc1`, `cArgLoc2`. `iOffs` is the offset within that memory block. This offset allows you to point to a specific audio port's set of parameters or state entries or entry. For example, the offset might be calculated as such:

```
iOffs = kTotalParameterBlockSize * portNumWeAreSetting + kParameterXOffset
```

Remember that a single code segment is used for all Processes; but conveniently, MuSh handles the initial offsetting to the correct instance if `iIsInstRel` is `true`. You need to only concern yourself with providing the offset to the correct audio port and parameter as shown above. Next, `iData` is a pointer to the values that you wish to send to the DSP. `iLen` is the number of values.

```
void GetMem
       (int iArgLoc, long iOffs, SInt32* iData, long iLen, bool iIsInstRel=true)
```

This call has the same form as `SetMem()`; but, obviously, data is fetched from the DSP memory.

```
void GetAndClearMem
       (int iArgLoc, long iOffs, SInt32* iData, long iLen, bool iIsInstRel=true)
```

This call retrieves data from the specified memory locations, then zeroes those locations. This is useful for retrieving meter values: The DSP code can accumulate a maximum value for the metering over the period of time between successive `GetAndClearMem()` calls. This call then resets the meter value to zero to restart the process.

```
void SetMemToConst
       (int iArgLoc, long iOffs, SInt32 iData, long iLen, bool iIsInstRel=true)
```

This call sets `iLen` number of memory locations starting at `iOffs` to the value `iData`.

```
void SetMemBuffered
              (int iArgLoc, long iOffs, int iBufArgLoc, long iBufOffs,
              SInt32* iData, long iLen, bool iIsInstRel=true)
```

There are some cases in which it is critical to update data, e.g. filter coefficients, synchronously with the audio sample interrupts. This call allows parameters to be placed in an intermediate DSP buffer, which will be updated synchronously within the next two sample periods. It is possible to specify an L memory location for this call; however, it is translated into two separate synchronous X and Y accesses.

```
void XetLMem(MuSh::EMoveType iMoveType, int iArgLoc, long iOffs,
              Cmn_Int64* iData, long iLen, bool iIsInstRel=true)
```

By specifying either MuSh::eSet or MuSh::eGet, L memory read and write accesses can be performed. This call operates similarly to `GetMem()` and `SetMem()`, except 64-bit host values are needed to hold the full 48-bit DSP value.

```
bool MuShReady() const
```

Calls to talk to the DSP (like the functions above) should be conditional based on the result of a call to MuShReady(). This will ensure that the MuSh system is ready to service your request. It is especially important to do this in MIDI MuSh plug-ins, since MIDI is in a separate thread and a MIDI plug-in's callback may interrupt the MuSh system while it is in an unstable state. See the SampleClick plug-in for an example of how to use this method.

Measuring and Verifying MuSh Cycle Counts

The MultiShell system provides some diagnostic support for measuring the cycle count usage of your MuSh plug-in. To enable this functionality, you must first create a text file named "`MuShOpts.txt`." This file should be peer with the Pro Tools application file. Within this file, add the lines:

```
UseLog 1
UseVerboseLogging 1
NumStatsPerActivate 10
```

After launching Pro Tools and instantiating the first MultiShell plug-in within a session, DSP Manager should create an acknowledgement file named "`MuShOpts.txt.ack.txt`." Within this file you should see the keywords and values repeated, as an acknowledgement that DSP Manager has correctly parsed them from `MuShOpts.txt`.

More importantly, a file named `MuShLog.html` should be generated. This will also be peer with the Pro Tools application file. Within this file you should find the cycle count information logged after a MultiShell plug-in has been instantiated. Note that for statistics to be gathered the "System Usage" window must be open, and Pro Tools must have focus.

To see an example, instantiate the Signal Generator plug-in. Statistics will be gathered one or more times and logged. If using an HD card, the resulting line in the log file should appear something like:

- `Gershwin slot 0 chip C: 0 = 1632 predicted - 1632 measured. predicted = 214 real work + 1418 NOPs.`

Translated, this means:

"We're on a Gershwin card in the first slot, on the 3rd chip on this card. The total number of cycles measured for the TDM interrupt was 1632, and the total predicted was also 1632, so the difference is 0. Of the 1632 cycles predicted, 214 come from the actual MuSh plug-in(s). The remaining 1418 cycles are comprised of NOPs inserted by MultiShell."

There are a couple of things to note about this measurement system. First, MultiShell is measuring and reporting its entire TDM interrupt cycle count; so, the value of 214 cycles isn't solely from the Signal Generator plug-in. Secondly, the measurements have a resolution of 2 cycles.

What's most important is that the resulting zero difference between the predicted and measured cycle counts indicates that cycles have been correctly reported by the Signal Generator plug-in (at least for one a single instance.)

Inserting multiple instances of the Signal Generator shows an increasing negative difference, i.e. MultiShell is measuring more cycles than as reported by SignalGenerator. This indicates that the cycle counts for the inner process loop may be wrong. However, this cumulative error is low and is hopefully masked by the additional cycle count headroom allocated by MultiShell.

Non-MuSh TDM Development

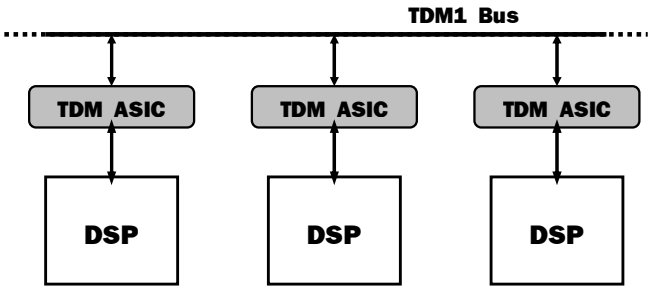
Advice for New 56k Developers!

It is highly advisable, if the MuSh architecture is compatible with your algorithm, that you do all initial development using MuSh, whether or not you intend to use MuSh in your end product. Using MuSh will allow you to get up and started faster by insulating you from some of the tedious details of 56k development. More importantly, the 56k chip apparently has a bug related to its "Host Port" mechanism. This bug causes the timeouts on the PCI bus and results in hard system crashes. The bug is triggered when communication gets out of sync between the DSP and the host code. By using MultiShell, with its standardized `GetMem()` and `SetMem()` API, this problem is much less likely to be encountered.

TDM1 Bus Architecture

This description of the TDM1 bus architecture is included as background information. HD systems use the TDM II bus architecture. The differences in this upgraded architecture are mentioned on page 105.

Each DSP in the TDM system is attached to a TDM ASIC (Application Specific Integrated Circuit). Since the ASIC is attached to the DSP's I/O bus, it appears to the DSP as normal block of memory space. These ASICs are then all connected together via the TDM bus. The following figure represents a simplified view of the TDM system involving three DSP chips.



The TDM bus then allows every DSP connected in a TDM system, even across multiple cards, to communicate with each other. The communication process occurs in sets, synchronized to the sample clock. Each sample period set is divided in 256 *timeslices*. During any of these timeslices it is possible for a single TDM ASIC to place a sample onto the TDM bus. Simultaneously, any number of other TDM ASICs can listen and capture this broadcasted sample. Hence, this is the origin of the name, time-division multiplexing (TDM).

Card Channels and Timeslices

For a DSP to access one of these timeslices, it simply reads or writes from a particular address of the TDM ASIC. Each memory address, which directly maps to a timeslice, is referred to as a *card channel*.

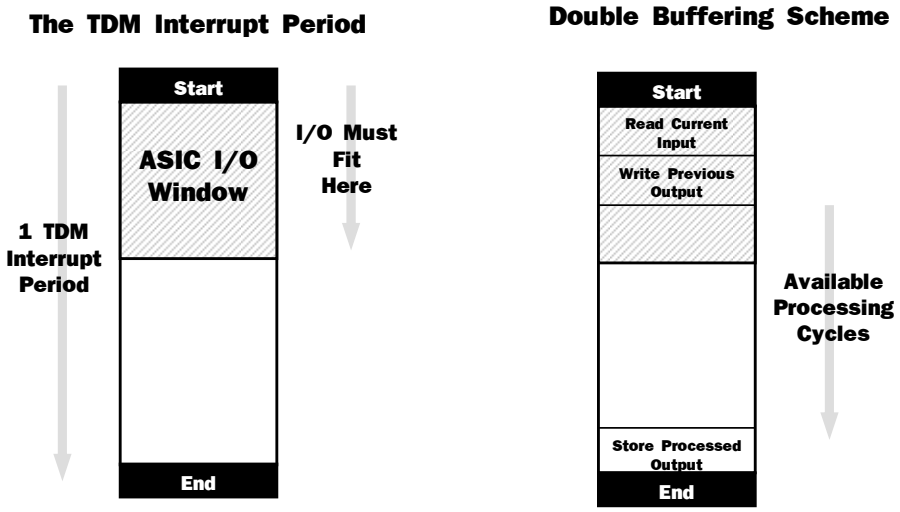
Loading DSP Code

When a TDM plug-in is first instantiated, its DSP code needs to be loaded onto an allocated DSP. Under the standard TDM development paradigm, a single DSP code binary is used for the entire plug-in -- for all of its Types. Therefore, this DSP code, which is stored in the Resource Fork on Mac, or the `.rsrc` section

for Windows, is specified at the Group level using the Effect Layer's `DefineDspResourceAndMaxChannels()` call. See the Appendices for usage. The most critical portion of the DSP code is the TDM interrupt routine, which is discussed next.

The TDM Interrupt

The DSP receives a hardware interrupt request at each sample period so that it can handle its I/O with the TDM ASIC. Since the ASIC has a fixed setup time requirement to meet the next set of TDM transactions, it's necessary to do all I/O as soon as possible within the interrupt. This enforces that a double buffering method be used. At the beginning of the TDM interrupt, all the processed output samples from the previous sample period are written to the ASIC. Then, the fresh incoming samples are read, or vice versa. After the TDM I/O is complete the remaining available cycles are free to implement the signal processing. This of course inherently introduces at least a one sample of delay in the processing.



Block/Background Processing

Certain algorithms require a large block of samples to be collected before the processing can be performed, e.g. an FFT. In addition, this sort of algorithm doesn't easily fit within the time constraints of the TDM interrupt period. Such an algorithm can operate in the "background" loop of the DSP code. The TDM interrupt is then just used to collect and output samples. The "background" loop waits for enough samples to fill up a buffer, then performs its algorithm.

Channel Allocation System

The Plug-In Library and the Effect Layer make use of the "channel allocation system." It is a development scheme that is well suited for plug-ins that have only a single Type, or very similar Types, which are capable of multiple instantiations on a single DSP. However, take note that the Channel Allocation System isn't a hard-coded standard for developing TDM plug-ins. Since the source of the Plug-In Library is provided in the SDK, the "system" can be tailored or radically altered to meet the requirements of your plug-in.

Under the channel allocation system, the time and space resources of a DSP can be thought of as being subdivided into a fixed and limited number of *channels*. (Unfortunately, this nomenclature clashes with the previous *card channel* term.) Through each of these channels audio data is streamed and processed. More specifically, a block of memory is reserved for each channel which is used to store parameters and state information. In addition, each channel requires a time slice of the DSP's processing power, which is utilized during the TDM interrupt.

An important precept of the channel allocation system is that each channel is processed under the same algorithm, using distinct control and state information. Herein lies the channel allocation system's biggest strength and greatest weakness. This uniformity greatly simplifies development, both on the host and the DSP, since accessing the channel's parameters and state is a standard procedure and only a matter of offsetting to the correct channel. On the other hand, the channel allocation scheme is poorly suited for algorithms that need to operate simultaneously on multiple and varied inputs and outputs, for example something like a surround mixing plug-in.

For a more concrete example of a plug-in well suited for the channel allocation system consider a EQ plug-in with both a mono and stereo Effect Type. The mono plug-in instantiation would require a single channel. A stereo plug-in would need two channels for each instance. Since, the stereo version would function more or less like a dual-mono there is little to differentiate these two Types.

Ports

In the TDM system, the inputs and outputs of a single instance are referred to as *ports*. Ports are enumerated starting at 1. For example, in a stereo plug-in, the “Left” port would be port number 1, and the “Right” would be port number 2.

Note that for the SetBypass function, 0 is a valid argument for the port number. In this case, the DAE client application (Pro Tools) is notifying the plug-in that all ports should be turned off at once.

The allocation of channels is handled by the Effect Layer, and provides the mapping of port numbers into channel numbers. The method `CEffectProcessTDM::GetChannelNumFromPortNum()` will convert a port number into the underlying DSP channel number. The number of channels allocated for a Process is equal to either the number of input or number of output ports, whichever is larger.

Channel Offsets

Unfortunately, there is no way to ensure that all the ports of an instance will be allocated on adjacent channels. As different instances with varied port sizes are inserted and then removed, the usage of channels will become fragmented. This is examined in better detail later.

However, some effort has been made in the channel allocation scheme to provide coupling between all the channels, or ports, belonging to a particular Process. This is done by creating a table of *channel offsets* in the DSP. Each channel has an offset entry in this table; this offset points to the next channel which is coupled to the current one. An offset of zero implies that this is the last channel, or port, of the current Process. The Effect Layer handles the creation of this table on the DSP in its `SetDSPInfo()` method. Of course, the DSP code must be written to handle this functionality.

See the latest Microbe TDM sample plug-in, which utilizes the Effect Layer, for an example of the channel offset method.

DSP Data Structures

Now, putting it all together, we see that there needs to be a minimum of six data structures stored on the DSP in the channel allocation system. (The following figures also show a 7th state block that an algorithm might happen to utilize.) These structures are represented in the figures below.

Card Channel Tables The input and output card channel tables are pointers to the TDM ASIC. If a channel is not connected to TDM, i.e. the channel is currently unused, then the addresses of the input and output table are made to point to an arbitrary "always zero" memory location, and a "bit bucket" location, respectively. This effectively turns the channel off, and passes zeroed samples into the algorithm.

Sample Tables The input and output sample tables store the incoming and outgoing samples, which are read and written using the input and output card channel tables.

Channel Offset and Parameter Tables After I/O has completed, the algorithm can operate, iterating through each channel and its associated set of parameters (and possible state information). The channel offset could optionally be used to determine which channels are associated ports.

The Microbe plug-in illustrates a possible implementation utilizing the channel offset table to handle stereo inputs.

Representation of all DSP data structures required for the Channel Allocation System

Input Card Channels
channel 0
channel 1
channel 2
channel 3
...

Output Card Channels
channel 0
channel 1
channel 2
channel 3
...

Input Audio Samples
channel 0
channel 1
channel 2
channel 3
...

Output Audio Samples
channel 0
channel 1
channel 2
channel 3
...

Channel Offset Table
channel 0
channel 1
channel 2
channel 3
...

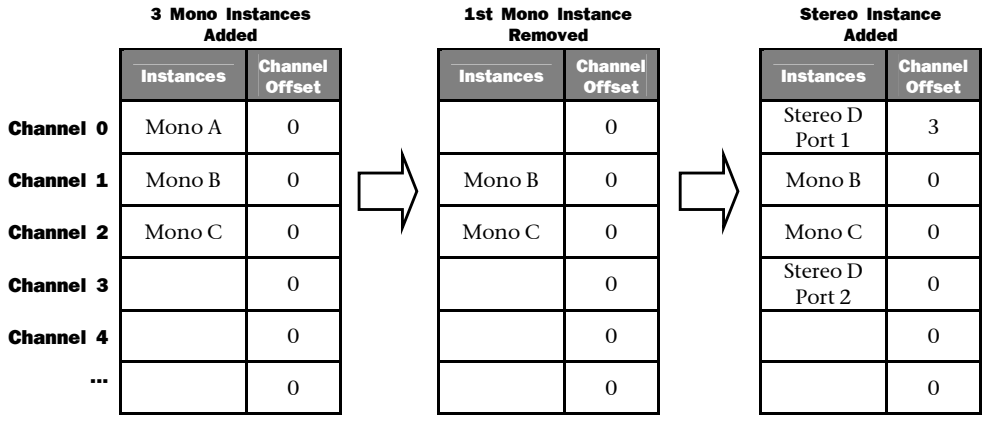
	Channel Parameters
Channel 0	parameter 1
	parameter 2
	parameter 3
Channel 1	parameter 1
	parameter 2
	parameter 3
Channel 2	parameter 1
	parameter 2
	parameter 3
Channel 3	...
	...

	Channel State Information
Channel 0	state variable 1
	state variable 2
	...
Channel 1	state variable 1
	state variable 2
	...
Channel 2	state variable 1
	state variable 2
	..
Channel 3	state variable 1
	state variable 2

Channel Allocation at Work

The following tables demonstrate how the channel offset table helps link the ports of an instance together. The first column of each table represents the conceptual channels running on the DSP and the instance that occupies each. The second column shows the configuration of the physical channel offset table that would be stored in DSP memory.

Initially the offset table is initialized to all zeroes. Then, three mono instances of a plug-in are added to the TDM session. The channel offsets of each are set to zero since there is only a single port for each. Next, the first instance is removed, leaving channel 0 empty. Lastly, when a stereo Type happens to be instantiated, its ports are split since the channel zero is open but the following is not. Therefore, the channel offset of port 1 is set to three -- the offset needed in the table to reach port 2. Port 2's channel offset is set to zero since it is the last port.



Host to DSP Communication

Host Commands Virtually all communication with the DSPs is initiated by one of the 56K's host commands. Each host command triggers a different interrupt within the DSP. The associated interrupt handling routine in the DSP code is then programmed to deal with that host command. Within this interrupt routine, additional data can be sent or received via the host port, coordinated with the C++ host-side code.

There are two Process level methods where DSP communication is sure to originate. They are `SetDSPInfo()` and `UpdateControlValueInAlgorithm()`. `SetDSPInfo()` is called immediately after the DSP code has been loaded and execution has begun, but before any input or output connections have been made to the Process. So, here you are allowed to initialized all your controls and other DSP state information. `UpdateControlValueInAlgorithm()` is called during runtime as controls are manipulated within the plug-in; obviously, as this method implies, it is used to update the parameters of your DSP Process.

The Effect Layer implements the two most basic and common methods for passing data back and forth between the host and DSP code. These are `CEffectProcessTDM::SendControlValueToDSP()` and `CEffectProcessTDM::GetValueFromDSP()`. Both send or receive a single 24-bit word (sign extended to a create a host 32 bit word) using a specified interrupt host command on a specified port number. The methods internally handle the conversion to the correct channel. Refer to the Appendices for more detailed usage.

Functions for Directly Accessing the DSP Host Port

If you need more flexible or complex DSP communication, you have full access to the `CXXXChipDSP` object (which derive from `CProcessDSP`) via the Process level `GetCProcessDSP()` method. Alternatively, you can access the `fOurDSPObject` member variable. This object, which represents the physical DSP chip, has methods to communicate with the DSP code. They are `DoHostCommandOnEmptyHostPort()`, `DoHostCommandWithData()` `SendLoLong()`, `SendLoLongBatch()`, `GetLoLong()` and `GetLoLongBatch()`.

The following functions were added or updated in Pro Tools 7.0. For reference, please refer to the function declarations in `CDSP.h`.

`SendLoLong()`: This function sends a value immediately to the FIFO on the DSP host port, without waiting until it has been flushed by the DSP, under the assumption that the buffer has free space. This can be called multiple times before issuing a `DoHostCommand()` call, however

SendLoLongBatch() provides the same behavior optimized for the different FIFO lengths of each different DSP chip. The state of the host port FIFO buffer may also be checked by the DoHostCommandOnEmptyHostPort() function. In Pro Tools versions before 7.0, SendLoLong() checked the host port FIFO to make sure it was empty before writing, which was safer but slower. This is no longer the case, and SendLoLong() writes immediately to the host port FIFO, as described above.

DoHostCommandOnEmptyHostPort(): This function has been added as a supplemental function to DoHostCommand(). It checks to make sure the Host Port is empty before sending a command to the DSP. This is necessary because SendLoLong() no longer checks the state of the host port before sending a value. If you are seeing DSP timeouts or values in the DSP host port FIFO being overwritten, you should change any use of DoHostCommand() in your plug-in to DoHostCommandOnEmptyHostPort(). Also, please be aware that all SendLoLong() calls should happen after the DoHostCommandOnEmptyHostPort() call, in order to ensure that the FIFO is not full before these values are written.

DoHostCommandWithData(): This function provides an efficient batch method for first executing a command on the DSP, then sending values to the DSP, and finally receiving return values from the DSP. Since these operations are so commonly called together, this function can more efficiently manage the state of the host port FIFO, regardless of the DSP chip on which a plug-in is being run. The parameters are as follows:

short	theHostCommand	command for the DSP to run
long	theGetCount	number of values you wish to receive from the DSP
long*	theGetData	buffer to store the received values
long	theSendCount	number of values you wish to send to the DSP
....		arbitrary-length list of values to be sent to the DSP, one value per parameter

SendLoLongBatch(): This function is a batch method of copying multiple values to the DSP host port FIFO. It is optimized for the FIFO queue length on each specific DSP chip. It is highly recommended you use this function in place of multiple SendLoLong() calls.

GetLoLongBatch(): This function is similar to SendLoLongBatch(), except it is a batch method of receving multiple values from the DSP host port FIFO. It is optimized for the queue length on each specific DSP chip. Additionally, it is not required to notify the DSP after each value is received. It only reports to the DSP after all values have been received, providing even greater efficiency.

Host Flags Another method of communicating with the DSP is via Host Flags. These are simple single bit uni-direction flags. There are a total of six on the 563xx architecture. The other three are controllable from the DSP core and readable by the host (HF3 through HF5). Three are controllable by the host and readable by the DSP (denoted HF0 through HF2). These flags do not trigger interrupts on the DSP and must be polled to see if they have changed.

The CProcessDSP methods used to manipulate and read the HF0-HF2 bits are:

	HF0	HF1	HF2
Set	SetHostFlag0()	SetHostFlag1()	SetHostFlag2()
Clear	ClearHostFlag0()	ClearHostFlag1()	(just not implemented.)
Get	Boolean GetHostFlag0()	Boolean GetHostFlag1()	Boolean GetHostFlag2()

Likewise, for HF3-HF5 the methods are Boolean GetHostFlag3(), Boolean GetHostFlag4(), and Boolean GetHostFlag5().

From the DSP side, it useful to use the bit manipulation/test opcodes to change the host flags. Some examples follow:

Bit Test and Set:

```
bset      #M_HF5,x:>M_DCTR    ; set host flag 5
```

Bit Test and Clear:

```
bclr      #M_HF5,x:>M_DCTR    ; clear host flag 5
```

Branch if bit Clear:

```
brclr     #M_HF1,x:>M_DSR,_HF1IsClear
```

Branch if bit Set:

```
brset     #M_HF1,x:>M_DSR,_HF1IsSet
```

PT6.0 or Later: MIDI & MIDI Event Host to DSP Communication

On Mac OS X or Windows XP, the plug-in's MIDI event handler function is called by Pro Tool's DirectMidi engine, which is running in a separate high priority thread. This is unlike OS 9, where the event handling is done at interrupt level.

OS 9: MIDI & Interrupt Level Host to DSP Communication

In Mac OS 9, MIDI data is delivered to plug-ins at interrupt level. Therefore, it is possible that the host-side plug-in code will be in the middle of a communication sequence with the DSP, when a MIDI interrupt is triggered. The MIDI interrupt handler will also likely need to issue host commands to the DSP code. However, in this scenario, the host command will need to be deferred until after the current host command is processed.

The Effect Layer's `SendControlValueToDSP()` method implements the necessary locking and buffering mechanisms to handle interrupt level communication. You should use it as a template for any custom communication routine used for a MIDI-enabled TDM plug-in. The relevant code pieces

```
ComponentResult CEffectProcessTDM::SendControlValueToDSP (long portNum, long hostCommandNum, SInt32 value)
{
     short channelNum = this->GetChannelNumFromPortNum(portNum);
     PIASSERT_NOTHROW(channelNum >= 0); // Assert channel allocation conversion succeeded.

     if ((fOurDSPObject != NULL))
     {
          // This locking/pending mechanism protects from MIDI interrupts attempting to do a host command while
          // non-interrupt priority code is also amidst a host command.
          if (fOurDSPObject->Lock() == true) {
               fOurDSPObject->DoHostCommand(hostCommandNum);
               fOurDSPObject->SendLoLong(channelNum);
               fOurDSPObject->SendLoLong(value);
               fOurDSPObject->Unlock();

               // We must lock the pending queue, so this codes not interrupted and manipulated.
               if(this->HostCommandPendingQueueLock())
               {
                    if (mPendingHostCommand_HeadIndex != mPendingHostCommand_TailIndex)
                    {
                         SInt32 pendingCommandNum = mPendingHostCommand_Num[mPendingHostCommand_TailIndex];
                         SInt32 pendingCommandPort = mPendingHostCommand_Port[mPendingHostCommand_TailIndex];
                         SInt32 pendingCommandValue = mPendingHostCommand_Value[mPendingHostCommand_TailIndex];
                         mPendingHostCommand_TailIndex =
                              (mPendingHostCommand_TailIndex + 1) % EffectLayerDef::MAX_PENDING_HOST_COMMANDS;
                         this->HostCommandPendingQueueUnlock();

                         SendControlValueToDSP(pendingCommandPort, pendingCommandNum, pendingCommandValue);
                    }
                    this->HostCommandPendingQueueUnlock();
               }
          }
          else
          {
               // This must be a MIDI interrupt, so lets add it to the list of pending host commands.
               mPendingHostCommand_Num[mPendingHostCommand_HeadIndex] = hostCommandNum;
               mPendingHostCommand_Port[mPendingHostCommand_HeadIndex] = portNum;
               mPendingHostCommand_Port[mPendingHostCommand_HeadIndex] = value;
```

```
                mPendingHostCommand_HeadIndex
                     = (mPendingHostCommand_HeadIndex + 1) % EffectLayerDef::MAX_PENDING_HOST_COMMANDS;

                return kDSPLocked;
           }
      }

      return noErr;
}


bool CEffectProcessTDM::HostCommandPendingQueueLock(void)
{
      bool result = false;

      if (mHostCommandPendingQueueLock == false)
      {
           mHostCommandPendingQueueLock = true;
           result = true;
      }

      return result;
}


void CEffectProcessTDM::HostCommandPendingQueueUnlock(void)
{
      mHostCommandPendingQueueLock = false;
}
```

Within the constructor, the following initializations are made:

```
      : mHostCommandPendingQueueLock(false),
      mPendingHostCommand_HeadIndex(0),
      mPendingHostCommand_TailIndex(0)
```

Sample Delay in TDM

As with any digital processing system, some amount of delay is incurred by the plug-in in the TDM system. The amount of this delay depends on the size and complexity of the processing algorithm.

In order to compensate for this delay, the method `GetDelaySamplesLong()` at the Process level should be overridden to return a value with the actual number of samples of delay your plug-in incurs. The DAE application or the user could then uses this value to adjust play lists and tracks in order to avoid phasing problems. It is very important that your plug-in accurately report the number of samples of delay so that the Automatic Delay Compensation feature in Pro Tools (6.4 or higher) functions properly. Please see further discussion of this method in the "Automatic Delay Compensation" section of the **Plug-In Features** chapter.

DSP Code Start Sequence

1 The plug-in DSP code is loaded using the Standard Shell which is preloaded on the DSP.

2 The host command `hcInitialize` is issued. This routine typically does the following things:

- Disables the TDM interrupt (IRQA) and DMA interrupts.
- Initializes the Modulo registers.
- Initializes the all input TDM card channel table entries to point to a empty memory location.
- Initializes the output card channel entries to point to a single "dead-zone" memory location.

3 After `hcIntialize` has completed, `hcStartDSP` is issued. There is a 0.5 second timeout period for the sending of any host command, pending completion of the previous. Therefore, the `hcInitialize` routine is allowed ample time to complete. This routine completes the following:

- Resets the stack pointer.
- Enables TDM interrupts (IRQA) to level 2 priority.
- Ensures that the host port priority level is enabled and set to level 0.
- Enables host command interrupts.
- Resets the interrupt priority level in the status register to level 0.

4 If the Process implements background loop processing, this can commence at the end of the initialization in `hcStartDSP`. Otherwise, if all processing is done in the TDM interrupt, start spinning in an infinite loop.

5 The Process level method `SetDSPInfo()` is invoked.

6 If or when the plug-in needs to be connected to the TDM bus, the necessary plug-in Library calls `ConnectInput()` and `ConnectOutput()` will be invoked to relay the card channel information to DSP code via the TDM host commands `tcConnectInput` and `tcConnectOutput`. These routines insert the card channel into the appropriate entry of the connection tables.

Mandatory Host Commands

At the minimum, there are three host commands that need to be implemented in the DSP code. They are outlined in the following table. Their implementation is mostly standardized and can be copied directly from the sample plug-in's `.asm` files.

Defined In StdOnyxDefs.asm StdPrestoDefs.asm	Onyx Interrupt Vector	Presto Interrupt Vector	Standard ASM Name	
`HostCmdInt0`	`$74`	`$94`	`hcInitialize`	Initializes DSP data structures
`HostCmdInt1`	`$76`	`$96`	`hcStartDSP`	Enables TDM interrupts
`HostCmdInt13`	`$8E`	`$AE`	`hcTDMCommand`	Handles TDM I/O connections to ASIC.

In addition, when using the "Channel Allocation System" of the Effect Layer, there are one or two more host commands that are required.

Defined In StdOnyxDefs.asm StdPrestoDefs.asm	Onyx Interrupt Vector	Presto Interrupt Vector	Standard ASM Name	
`HostCmdInt6`	`$`	`$`	`hcSetNumChannels`	Gershwin only. Sets the maximum number of channels to process for the current sampling rate.
`HostCmdInt7`	`$`	`$`	`hcSetChannelOffset`	Used to create the offset table used in the Channel Allocation System.

The following section on the Standard Shell reiterates some this information for your reading enjoyment.

The Standard Shell and SADriver, a.k.a. "Sad River"

The Standard Shell is responsible for loading your DSP code onto an available DSP. This is done using the SADriver routine, `LoadShell()`, which is called in the function `CDSP::Load()`. In general, three main SADriver routines are involved in starting the DSP plug-in code running: `LoadShell()`, `InitShell()` and `StartShell()` (however, `InitShell()` and `StartShell()` must be implemented by the SADriver client to do anything useful; which is exactly what the plug-in Library does). Although it is not necessary to know all the details for plug-in development, here are the general actions of these SADriver calls:

◆ `LoadShell()` loads the plug-in's DSP code onto a previously allocated DSP. The sequence is as follows.

1 The Standard Shell will be loaded if the previous shell was not the Standard Shell.

2. `LoadShell()` will then load the plug-in's DSP code. `LoadShell()` makes sure that the DSP's host port transmit register is empty by reading any data that may be coming from the DSP.
3. `LoadShell()` assigns the Shell (which for a plug-in are routines dispatched via the `DSPShellDispatcher()` in CDSP) to the DSP and calls `InitShell()` for the DSP (where "the DSP" is a CDSPobject).
4. `LoadShell()` uses `ChangeXBits()` to setup the SSI interrupt enable bits for the given card.
5. `LoadShell()` calls `SetLeftRightBit()` with the appropriate value for the type of card. If the previous shell was not the Standard Shell then `UpdateDSPPeripheral()` is called for the DSP.
6. `LoadShell()` calls the `StartShell()` routine for the DSP.

◆ The plug-in library function `InitShell()` calls the `hcInitialize` vector in your DSP code. Use the `hcInitialize` section of your DSP code for any required initializations. For instance, be sure to reset all the M registers as you need them; that is, never assume the DSP has been left in any particular state prior to use by your own DSP code.

◆ The plug-in Library function `StartShell()` calls the `hcStartDSP` vector in your DSP code. This is called next after `InitShell()` and is used to start your DSP code running. This is where you can do any desired background processing.

Note: the SADriver routine `ResetDSP()` (which forces a hardware reset of the DSP), will in general, not be called. Therefore, you should not rely on a hardware reset for any type of plug-in reset or initialization.

Note also, the Standard Shell does more than just load code. In addition to the tasks above, it is responsible for communicating with and configuring the TDM hardware, DigiShift hardware, and allows DSP Probe to examine the DSP contents.

Execution of Standard Shell routines on the DSP is done by performing an `NMI` host command. The vector for this routine is at `P:$001E` (and `VBA:$0A` for Onyx). The Standard Shell DSP code uses the `R3`, `N3`, `M3`, `X0`, `X1` and `B1` registers. The DSP Standard Shell saves these registers before they are used and restores them when it is finished. This is important to know because if you use any higher priority interrupts, you will need to save and restore these registers before using them. However, plug-in developers should not need to be concerned with this aspect, since plug-ins run at a lower priority level than the NMI. Therefore, all registers are available for plug-ins to use.

Please refer to the specific memory maps for the placement of the standard shell, so that you can avoid overwriting this important piece of code when your own DSP code is loaded by `LoadShell()`.

DSP Manager

Note: This section focuses on the aspects of Non-MultiShell usage.

The DSP Manager was developed to manage and optimize DSP resources within the TDM system. This management is made necessary by the varying ram configurations of different DSPs.

All TDM plug-ins interact with the DSP Manager, via underlying mechanisms of the plug-in Library, to activate and deactivate themselves on DSP chips. The DSP Manager is a shared library that the plug-in links to at runtime. When a TDM plug-in process instantiates, it calls the DSP Manager and registers itself, and then requests that the DSP Manager activate the DSP process. After allocating a DSP chip, the DSP Manager is then able to initiate the process. The DSP Manager is also capable of deactivating processes from DSPs. If no DSP is available to activate a process on, the DSP Manager can optimize DSP usage by deactivating all processes currently activated, sorting them by a deterministic algorithm, and reactivating them in the order created by the sort algorithm. The DSP Manager is also capable of doing a 'pre-flight' check as to whether a set of processes can be activated by virtually activating processes on virtual DSPs.

Additional DSP Manager Information

These DSP Manager capabilities allow it to optimize DSP usage on-the-fly, allow plug-ins to use the DSP more efficiently when sharing a DSP with different process types, and guarantee that any session created by the user will re-open on a system with the same DSP resources available. The functionality of the plug-in Library to allow the DSP Manager to move and optimize plug-in's DSP usage is embodied in the methods `CDSPProcessType::AllocateOpaqueDSP()`, `CDSPProcess::Deactivate()`, and `CDSPProcess::ActivateOnOpaqueDSP()`.

`CDSPProcessType::InstantiateProcess()` calls the DSP Manager to activate the process. When the DSP Manager activates a process, it calls `CDSPProcessType::AllocateOpaqueDSP()` to allocate a DSP chip, and then it calls `CDSPProcess::ActivateOnOpaqueDSP()` to tell the plug-in to start running on the DSP. If, later on, the DSP Manager needs to move the process to a different DSP, it will call `CDSPProcess::Deactivate()` to tell the process to suspend running on its current DSP before it activates it on a different DSP. The ramification of this is that the process data members `fOurDSPObject`, which can be accessed via the `CDSPProcess::GetCProcessDSP()` method, and `fOurDSPPtr`, which can be acquired by the `CDSPProcess::GetDSPPtr()` method, will be set to `NULL` while the process is deactivated and your plug-in must be careful not to reference these `NULL` values!

Digidesign Object Architecture

Interfacing to the DSP Manager is provided by the files contained in a folder called DOA (Digidesign Object Architecture), in the plug-in Library. This folder contains an implementation of the basics of Microsoft's Component Object Model (COM) along with some additions to make programming with it easier and less error-prone. By using COM to implement the binary interface between the DSP Manager and a plug-in, Digidesign gains the benefit of using a proven standardized design. Also by doing so, the goal is to ensure that in the future Digidesign can retain backward compatibility with plug-ins compiled with the DSP Manager that ships.

The DSP Manager consists of a shared library that ships with Pro Tools, and a static library (i.e., the DSPManagerClientLib compiled into the plug-in Library) for clients, which is essentially just a stub encapsulating the connection to the shared library.

The shared library provides a single function, `GetDSPManager()`, declared in `DSPManagerExports.h`. This returns a COM object that can be queried for various interfaces that the DSP Manager might support. Currently this object supports two interfaces, `IDSPManager` and `IMultiShell`. `IDSPManager` provides the functions relating to process activation and deactivation. `IMultiShell` is the interface necessary for a plug-in to support MultiShell.

The static client library provides type-safe functions to get each of the current interfaces supported by the DSP Manager. The client library takes care of the details of connecting to the shared library and acquiring and caching the requested interfaces. The client library needs to understand COM and shared libraries, but plug-ins linking to it do not. They simply ask for interfaces and then call functions on these interfaces. The client library weak-links to the shared library so any plug-in using it is automatically linked appropriately to the shared library.

The most important thing to know is that it is illegal to cast an `IProcessXXX` class or any of the newer COM interfaces. It is illegal even if you "know" that you are casting to the true type and even if you are using the C++ `static_cast<>` and `dynamic_cast<>` correctly. The reason is that plug-ins may receive references to objects created by the DSPManager or by different plug-ins and the cast may not be valid in the version of the code base used to compile these. Even the implementation of dynamic cast and run-time type identification may have changed.

Inserting a Plug-In: The DSP Manager at Work

This segment briefly outlines the relationship between the DSP Manager and other plug-ins. This is intended to be a general overview, and not a detailed account. You should already have a basic understanding of the existence of Virtual and Physical DSP allocation on the TDM Platform, shown in the previous sections. The virtual allocation system allows the TDM system to rapidly determine optimal plug-in placement prior to allocation of physical resources. Virtual allocation can save a great deal of time when plug-in shuffling is required.

Instantiation

The first thing that happens during plug-in instantiation is that the hosting application calls into the plug-in with `PI_InstantiateProcess()`. This call will eventually filter down to the process type level through `CDSPProcessType:: InstantiateProcess()`. At this level, we begin the Activation process, which consists of Virtual and Physical resource allocation.

There are two important calls that need to be made back to the DSP Manager. The first call is `RegisterProcess()`, which alerts the DSP manager to register the plug-in's process type. The second call is `ActivateProcess()`. During the call to `ActivateProcess`, all of the virtual and physical allocation occurs.

Process Activation

Each time a plug-in calls upon the DSP Manager for activation, it responds by doing a pre-flight check of the system. The DSP Manager needs this check to determine if there are enough resources to instantiate the new plug-in. This is the reason for using the virtual allocation system. During this pre-flight check, the DSP Manager will put the plug-ins into a prioritized list and assign virtual resources to them. The following criteria are used for prioritizing plug-ins.

- Number / Types of TDM hardware that the plug-in supports (MIX / Merle/ Nubus)
- Number / Type of DSP cores supported (MIX hardware only).
- Number of DSP cycles plug-in uses per TDM interrupt

The following methods are used by the DSP Manager in order to gather information needed for prioritization, `GetDSPCoreTypeList()`, `GetDriverTypeList()`, `GetNumCoreTypes()`, `GetNumDSPCyclesPerProcess()`.

Virtual Allocation

Once the plug-ins have been prioritized, virtual allocation begins. The DSP manager will call back into each plug-in with `AllocateVirtualDSP()`. During this call your plug-in must determine if the new process will fit on the available assigned virtual resources. If the plug-in determines that there are no virtual resources available for the new process, then it will call back to the DSP Manager to request a new virtual DSP. It is the plug-in's responsibility (handled by the plug-in Library) to keep a list of assigned virtual DSPs, and how much space is available on them. Virtual allocation then continues for each plug-in in the DSP Manager's prioritized list.

If there are enough virtual resources to accommodate all plug-ins in the DSP Manager's list, then virtual allocation will be successful. Otherwise the virtual allocation will fail, and no physical allocation will take place. It is important to note that the DSP Manager can make several attempts at prioritizing plug-ins. Therefore, if the first attempt at virtual allocation fails, others attempts will pursue. In the event that all attempts at virtual allocation fail, the user will be confronted with an error dialog indicating that there are not enough resources to accommodate the request.

The physical allocation process begins immediately after the first successful attempt at virtual allocation. Physical allocation mimics the successful virtual allocation attempt. The only difference is that actual physical DSPs will be used instead of virtual DSPs. During the physical allocation, shuffling of plug-ins may occur.

The DSP Shuffler Plug-In

Available for download on the developer website is a plug-in called "Shuffler". It may be useful for debugging issues related to plug-ins having problems by being shuffled. To use, place it in your plug-ins folder. You will see a plug-in called "Shuffler" in the mono TDM insert list. Instantiate it and a shuffle will occur. No plug-in will actually ever be instantiated. It works by fooling the DSP Manager into believing that it will be able to find a DSP to run on, but it reports back that it can't actually find one. The DSP Manager, in a futile attempt to make room for the plug-in, performs a shuffle.

Optimal Plug-In Load Ordering On the DSPs

A plug-in can suggest to the DSP Manager which DSPs it would rather load on first, if there is a choice. For example, a plug-in may be able to load on both a Presto DSP and a Motorola 56321 DSP, but prefer to load on a Presto DSP in the case that both are available. This preference is suggested by the position of the card types and DSP core types in the DSP core type list and the driver type list. The types of higher preference are closer to the beginning of the lists.

If a plug-in has no specified DSP load order preference, it follows the default load order. ***In an attempt to utilize DSPs more efficiently and minimize reordering of plug-ins across DSPs, the default load order of preference for DSPs has been modified for Pro Tools 7.*** *Now plug-ins will be loaded onto the DSPs in the following order of preference: Presto, non-RAM 56321, and SRAM 56321. The load order was reversed when the HD\|Accel card was first released because few plug-ins had been ported to the new DSP (effectively, it preserved the original Presto DSPs). Now, since most plug-ins have been updated to support the 56321, we're reverting the load order to preserve the more powerful DSPs on the Accel card.*

The order of these lists can be modified in a plug-in's `GetDriverTypeList()` and `GetDSPCoreTypeList()` functions. For example, to prefer a Gershwin card over a Gershwin II card, code something analogous the following in `GetDriverTypeList()`:

```
long *longPtr = (long *) *typeList;
if (gershwinId) *longPtr++ = gershwinId;
if (gershwin2Id) *longPtr++ = gershwin2Id;
```

The Gershwin card is added to `typeList` before the Gershwin II card is. The Gershwin card is ahead in the list.

Likewise, to prefer the Motorola 56321 over the Presto, code something like this in `GetDSPCoreTypeList()`:

```
if (cardType == kTDM2GenericType)
{
	if((m321DspResource_SRAM != EffectLayerDef::NO_DSP_RESOURCE) ||
		(m321DspResource != EffectLayerDef::NO_DSP_RESOURCE))
	{
		types[(*numCoreTypes)++] = kTDM_321_SRAM_Core;
	}
	if(m321DspResource != EffectLayerDef::NO_DSP_RESOURCE)
	{
		types[(*numCoreTypes)++] = kTDM_321_NoSRAM_Core;
	}
	if(mPrestoDspResource != EffectLayerDef::NO_DSP_RESOURCE)
	{
```

```
                types[(*numCoreTypes)++] = kTDM_Presto_IO_SRAM_Core;
                types[(*numCoreTypes)++] = kTDM_Presto_SRAM_Core; //
        }
}
```

Here, for this particular card type, the 56321 core types are added ahead of the Presto core types in the `types` array, which represents the type list in this function. Note that the card type is determined before the core type and that the core type is determined with respect to given card type.

Using DMA with the TDM II ASIC

The DMA controller on 321 chips can be used to quickly and asycnhronously transfer large amounts of data between the host and the DSP. This method is described in Tech Note #18: DMA host port polling, found in the "Tech Notes' section of developer.digidesign.com.

DSP Probe Tool

The DSPProbe has not been ported to OS X. The DSP Debugger contains all the functionality of the former DSPProbe.

The Digi 56k DSP Debugger

The DSP Debugger is a TDM plug-in that can be inserted anywhere within a session. It then provides a debugging environment which can attach to another plug-in's DSP within that session.

Compatibility

- Mac OS 9, Onyx or Presto chips, PT v5.1 or later.
- Mac OS X, Onyx, Presto, or 321 chips, PT 6.0 or later.
- Windows XP and Mac OS X, Onyx, Presto, or 321 chips, PT 7.0 or later.
- OS X version is a Universal Binary

New Configuration

As of version 7.3, the DSP Debugger now consists of two separate components: the DSPDebugger.dpm plugin and a separate Disassemble56k executable binary. Both of these components bust be present in the plugin folder to work properly. Please install them together. Within the debugger UI, you must also supply the path to the Disassemble56k executable.

Setting Up

1 A block of assembly code that you wish to debug must first be wrapped with uniquely identifiable labels. Wrap a particular section of code in your `.asm` file with two labels before assembly. The default labels are "TdmTool_Interrupt" and TdmTool_InterruptEnd", however you can use any labels you like. For MultiShell, these labels are not needed since the entire DSP code fragment is loaded by default.

2 Put the debugger plugin and the separate disassembler executable into the "Plug-Ins" folder and launch Pro Tools. Insert the plug-in to be debugged and the DSP Debugger plug-in into a session. On Pro Tools|HD Accel systems, it doesn't matter if the plug-in to be debugged is on a Presto chip and the debugger is on a 321 chip, or vice-versa.

3 Switch to the "DSP Setup" page using the control in the lower right-hand corner.

4 Select the DSP that is hosting your target plug-in. Use the System Usage window of Pro Tools to find this information. If there are multiple cards in your system, you will see the cards labelled “Slot 0”, “Slot 1”, etc. They will appear in the same order as in the System Usage window. The ordering of the cards in the menu is more important than the numbering. For example, the menu may show “Slot 1 DSP 1, Slot 1 DSP 2... Slot 0 DSP 1, Slot 0 DSP 2...” where Slot 1 – because it is first in the menu – indicates the first card and Slot 0 indicates the second card. Make sure to not only pick the correct DSP, but also the correct card!

5 If you only need to probe memory address and are not in need of breakpointing functionality, click the "Only Probe" button, and skip to step number 9.

6 The `.lod` file produced by the 56k assembler for the target plug-in must be available. Click the arrow for the "DSP Code LOD File:" field. This will bring up a file finder. Browse to and choose your `.lod` file.

7 Similarly, click the arrow to supply the path to the Dissasemble56k executable. This should be in the “Plug-Ins” folder.

8 If the target plug-in uses MultiShell, enable the “MultiShell” button.

9 The Debugger must load a small bit of additional code onto the target DSP. The "Debugger Addr:" defaults to the top most region of internal P RAM. Change if needed. The length of this code is displayed when the Debugger is first started.

10 Fill in the "Start Label:" and "End Label:" boxes with the labels that you used from step 1 (if different from the default).

11 Switch back to the "Environment" page.

12 The columns on the right are memory watches. Enter memory addresses that you wish to watch during the running and debugging of your plug-in. There are several ways to enter addresses and effective address. (See examples below). Labels from your `.asm` file can be entered here and the debugger will acquire the actual address and memory space from the `.lod` file.

Example entries:

Label:	`"LeftReverbBuffer1"`
Hex:	`"x:$f62"`
Decimal:	`"y:894"`
R Register:	`"x:r4"`
Previous address watch plus one:	`"++"`

Note: You should set at least one memory watch for the debugger to work properly.

13 Do a "Save Settings" if you wish to retain this information in a settings file for later recall.

Usage

1 Press the "Play" button. This will load and parse the `.lod` file, and load the "trap code" onto the target DSP. Text of your disassembled DSP code should be displayed. Memory watches will now enabled and should continuously update.

2 Use the slider to scroll to the instruction you wish to break at. Click the breakpoint square to the left of the instruction. If this instruction is within program flow, the breakpoint should be encountered and the instruction address to the right of the breakpoint button will be highlighted. Additionally, all registers should be updated to reflect the current context at the breakpoint.

3 To modify a register or a memory address, type in a new value. Either hexadecimal or decimal are accepted. Prefix hex values with "0x". Status registers and multiple word registers are not mutable, e.g. registers X10, B210, MR.

4 To trace to the next instruction, click the "Foot" button. Otherwise, to resume program flow, click the "Play" button.

Other Stuff

To change the number representation of the registers or memory watches, click the yellow square to the right of the value. This will cycle between hexadecimal, decimal, and fixed-point.

Known Bugs

When debugging MultiShell on HD, audio is muted after "Play" button is pressed. Workaround: Instantiate a second process of your MultiShell plug-in. This will re-enable audio. Things seem to work okay otherwise.

Limitations

1 Can't use Host Flags 1 and 5 in your plug-in. (The debugger uses them.)

2 Only one section of code can be extracted from the `.lod` file.

3 Debugger can't decode all 56300 instructions. Unknown opcodes will be displayed as constants, e.g. `DC $053af2`.

4 Breakpointing the TDM interrupt or the "Background Loop" should work fine. Beware of breakpointing Host Commands interrupt handlers since you can potentially stall communication between the host-side C++ code and DSP code, resulting in timeout errors & crashes.

The Debugger and HCP Copy Protection

The DSP Debugger cannot be used with PACE's HCP protected plug-ins. The DSP Debugger uses shell commands to talk to the DSP and HCP repatches those. This is for the security of those using HCP encryption.

Part IV: Host-Based Processing

Chapter 9: RTAS And AudioSuite

The Plug-In Environment chapter discussed the AudioSuite and RTAS architectures in general. Now, let's point out a few functional details.

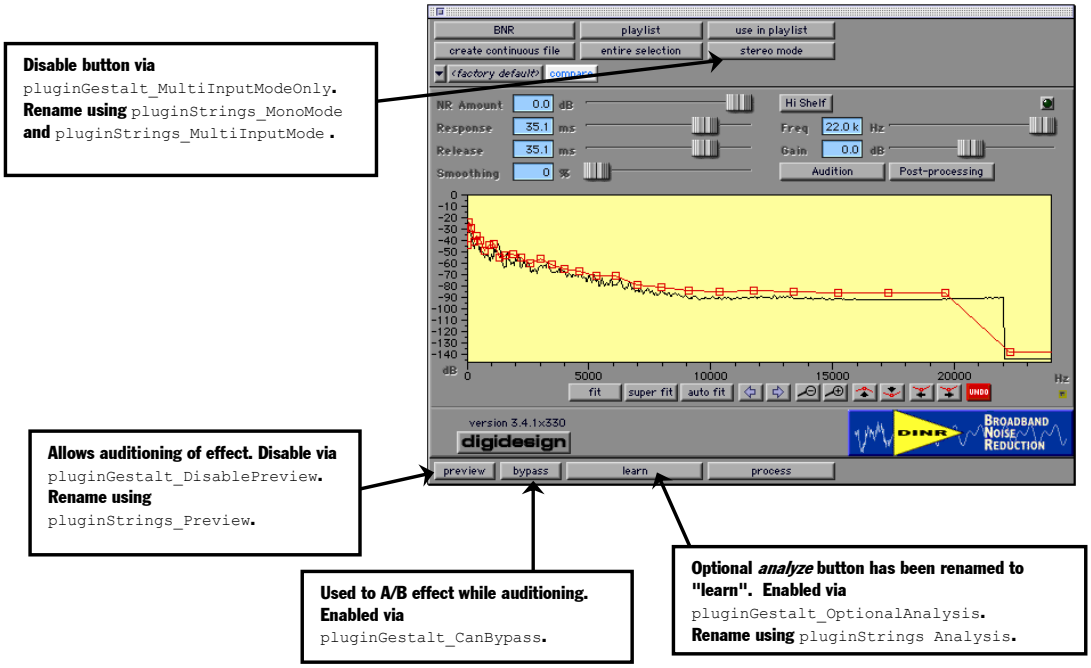
AudioSuite Options

In addition to AudioSuite's normal processing mode, there exists a Preview mode that allows the user to audition the effect in real-time or semi-real-time before it is committed to disk. This mode doesn't rely on any additional code within the plug-in, but uses the existing `ProcessAudio()` method already implemented. If your algorithm is not capable of real-time behavior, or is simply unsuited for this functionality, it is possible to disable the mode.

AudioSuite also allows for a pre-analysis pass of the audio data, immediately before the `ProcessAudio()` pass. This option can either be turned off, be enforced, or be optional to the user. When the optional mode is selected, DAE automatically adds an *analyze* button to the UI window.

Additionally, AudioSuite allows the developer to customize much of the button text displayed on the UI window which wraps an AudioSuite plug-in.

For the details of these options, see the **Plug-In Library/Effect Layer Reference Guide**. The following diagram attempts to present some of the options.



Sample Data Types

◆ Using the Effect Layer, the data type of AudioSuite samples are in the format ±1.0, 32-bit floating point. If necessary, by overriding `CEffectProcessAS::GetConnectionFormat()` and `CEffectProcessAS::SetConnectionFormat()` it is possible to process signed 32-bit integer samples instead (`dataType_32BitInt`).

◆ RTAS buffers always enforce the use of 32-bit floating point data.

Processing in the Callback

The AudioSuite architecture invokes the `ProcessAudio()` callback on large chunks of sample data, with a size dependent on the "AudioSuite Buffer Size" Preference within Pro Tools. RTAS, on the other hand, has its sample block size fixed at 32 samples.

Stem Formats

See the **Multichannel Plug-in Specification** document for more details on how the specification of a stem format affects the operation of an AudioSuite plug-in.

Prime()

The `Prime()` method is invoked with a `true` parameter before the start of an Analysis pass or a Process pass. A `Prime(false)` is issued at the end of both passes. This method gives the plug-in the opportunity to do any initializations before the processing actually begins. **Note: Prime() is also called on start and stop of the transport. This is because the Prime() callback is handled in the same manner for both RTAS and AS by DAE, and RTAS depends on Prime() being called prior to start of the transport in order to operate correctly.**

AudioSuite Non-Linear Capabilities

The generation of output samples by an AudioSuite Process must occur linearly and incrementally; however, the source of input samples is random access. Before every `ProcessAudio()` call, the method `TranslateOutputSampleNum()` is invoked by DAE. Here, the Process is given the opportunity to specify its required block of input samples given an output sample number. These input buffers are then appropriately filled before the `ProcessAudio()` call is made. A Type should alert DAE that it using non-linear functionality by defining the `pluginGestalt_UsesRandomAccess`.

The following data structure is available to an AudioSuite plug-in and represents a single channel of incoming or outgoing audio. Using the Effect Layer, the methods `GetInputConnection(long connectionIndex)` and `GetOutputConnection(long connectionIndex)` return the specified `DAEConnectionPtr`, which points to this structure.

```
typedef struct DAEConnection
{
     long               mConnectionType;         // Type of connection being made (pluginType_TDM,pluginType_ASP...).
     DAEConnection      *mNext;                  // For internal use only.

     // The following fields are TDM specific.
     Ptr                mTDMConnection;          // This is the old style of connection record.

     // The following fields are ASP specific.
     long               mBufferSize;             // Size of buffer in bytes.
```

```
    Ptr                 mBuffer;                // Location of buffer containing the data.
    long                mNumSamplesInBuf;       // Number of samples in buffer which have not yet been transferred.
    long                mStartBound;            // Start bound of all the samples passed through this connection.
    long                mEndBound;              // End bound of all the samples passed through this connection.

    SFicPlugInSpecPtr   mSourceSpec;            // Source of data going into the buffer.
    SFicPlugInSpecPtr   mDestSpec;              // Destination for the data.
    short               mOutputNum;             // Output number for the source data.
    short               mInputNum;              // Input number for the destination data.
    ConnectionFormat    mFormat;                // Format of the buffer data.
} DAEConnection, *DAEConnectionPtr;
```

`mStartBound` and `mEndBound` specify the first and last sample number that will be passed through this connection. These sample numbers correspond to what would be displayed in Pro Tools on the timeline or in the Event Edit Area.

`mNumSamplesInBuf` informs the process how many samples are currently present in the sample buffer pointed to by `mBuffer`. Since AudioSuite does its processing in blocks, these data members are appropriately updated before every `ProcessData()` or `AnalyzeData()` call.

InitOutputBounds

The `InitOutputBounds()` method is invoked in several different cases:

- Before an analyze, process or preview of data begins.
- At the end of every preview loop.
- Or, after the user makes a new data selection in Pro Tools.

This allows the AudioSuite process to adjust the `mStartBound` and `mEndBounds` members of the output connection(s) to vary the length and/or start and end points of the outputted audio region.

ProcessAudio

At the completion of the `ProcessAudio` callback, all input connection's `mNumSamplesInBuf` members should be zeroed. The number of samples generated for each output should be placed in the output connection's `mNumSamplesInBuf` member variable. In addition (redundantly), the number of output samples that were generated should be the return value of the `ProcessAudio()` call. If it's required that this value sometimes be zero, then the gestalt `pluginGestalt_DoesntIncrOutputSample` should be defined by the Type. This would be necessary if `TranslateOutputSampleNum()` needs to be called more than once for the same output sample number. This would allow the process to grab samples from more than one block to calculate the current output.

If the plug-in is capable of multi-channel processing, it should process all channels during each call to `ProcessAudio()`. As such, the preceding call to `TranslateOutputSampleNum()` will provide the same input sample number for all channels, rendering the portNum passed into `TranslateOutputSampleNum()` useless. In other words, a multi-channel-capable AudioSuite plug-in cannot get a block of input data for processing from different places in time. (Please see the Reverse Double Half plug-in as an example. For further information about multi-channel AudioSuite plug-ins, please read the AudioSuite section of the **Multichannel Plug-in Specification** document.)

RTAS Specifics

DAEConnection Structures

The `DAEConnection` record available to RTAS processes differs from that of an AudioSuite plug-in. In fact, only the data members `mBuffer` and `mNumSamplesInBuf` contain valid data within the structure.

Prime()

For RTAS, the `Prime()` method is invoked with a true parameter whenever the user starts the transport. A `Prime(false)` is issued when the transport is stopped. However, typically, the RTAS engine is never stopped. So, the `Prime()` method gives no indication to the state of incoming `ProcessData()` (or `ProcessAudio()` with the Effect Layer) calls.

RTAS Improvements

Since version 7.0 of Pro Tools, there are some major improvements that developers of RTAS plug-ins should be aware of:

Multi-threading for RTAS Processes

Support for multi-threaded RTAS processes is built into Pro Tools starting in version 7. Ideally plug-ins should not need to make any modifications for this as long as they are thread-safe, particularly in their processing code. In other words, RTAS plug-ins must be written such a way that multiple instances of the same plug-in can safely operate concurrently. They must not share non-static data in a way which produces failures or incorrect results if the code is re-entered in different threads. The SDK is set up so that each instance is represented by a separate class object - `CProcess` or `CEffectProcess`, for example, so many plug-ins are expected to be thread-safe without any additional modification. Plug-in instances will continue to see all processed samples one at a time in the correct order.

Variable RTAS Buffer Sizes

Starting in Pro Tools 7, plug-ins will be able to use larger RTAS quantum sizes than previously possible. In all previous releases, the quantum was always 32 samples in length. Now the RTAS quanta will be variable, with 32 samples as the minimum quantum size. For now, RTAS buffer sizes will vary based on the Hardware Buffer Size that the user sets in Pro Tools. This may be modified in the future.

The performance benefit of this feature is that plug-ins which spend a fixed amount of time doing per-call setup will have that time amortized across a much larger number of samples, resulting in decreased CPU requirements. Early testing indicated that some plug-ins spend around 50% (or more) of the time doing setup work when the quantum is 32-samples in size. By increasing the quantum from 32 samples to 1024 samples, the CPU usage of such a plug-in is cut virtually in half. Some plug-ins - like EQs, for example - may not have much per-call setup time at all, in which case the benefits will be less dramatic.

To take advantage of variable quanta, plug-ins will need to make the following modifications:

- Add a new gestalt: `pluginGestalt_SupportsVariableQuanta`

In the Effect Layer, this can be done via the `AddGestalt()` function. Plug-ins not supporting the Effect Layer will have to return true to this gestalt in their implementation of `Gestalt()`. The gestalt must be handled **AT THE TYPE LEVEL**.

A plug-in which returns true to this gestalt is telling DAE it can process data at any quantum size. When used with a version of Pro Tools/DAE not supporting this feature, the plug-in will receive 32 samples of data at a time.

A plug-in which returns false to this gestalt will always receive 32 samples of data at a time.

- The algorithm code in the plug-in must not contain any hard-coded quantum size values (i.e. 32), and must be able to handle any possible quantum size for its buffer.

  To aid in accomplishing this, the following new API is available for EffectLayer plug-ins:

  `CEffectProcessRTAS::GetMaximumRTASQuantum()` - Returns the largest quanta size DAE might ever pass to plug-ins.

  The value returned by this function DOES NOT CHANGE throughout the life of the plug-in (from instantiation to de-instantiation). This function should be used for plug-ins which need to allocate data structures based on the processing quantum instead of hard-coding any particular value. The function is backwards compatible with versions of DAE earlier than 7.0 and therefore adding it will not break your plug-in's compatibility with older versions of Pro Tools.

  Non-EffectLayer plug-ins should implement their own version of GetMaximumRTASQuantum() at the Process level, using CEffectProcessRTAS::GetMaximumRTASQuantum() as a model.

Frequently Asked Questions (FAQ)

Q: How do I know whether or not variable RTAS buffer sizes are supported in the version of Pro Tools my plug-in is being instantiated on?

A: There is a new Fic gestalt: `eFicGestalt_VariableRTASQuanta`. With this Fic gestalt, a plug-in can query the currently running DAE to see if it supports variable RTAS buffer sizes. To make the query, you'll want to do something like this:

```
long ficHasVariableQuanta = 0;
FicGestalt(eFicGestalt_VariableRTASQuanta,&ficHasVariableQuanta);
if (ficHasVariableQuanta != 0)
       useVariableQuantaInYourPlugIn = 1;
```

Q: How do I know what the current RTAS quantum is at any given time?

A: For newer true-RTAS implementations based on the `CEffectProcessRTAS` architecture, the current quantum size is passed in as the final argument to the `RenderAudio` member function of the RTAS plug-in's process class (`frames`).

For traditional AudioSuite-wrapped RTAS implementations based on the `CEffectProcessAS` architecture, the current quantum size is stored in the `mNumSamplesInBuf` field of the `DAEConnection` structure(s) pointed to by the `mInputConnection` member of the RTAS plug-in's process class. These are accessible via a call to `CEffectProcessAS::GetInputConnection()`.

Q: What if I am doing my own buffering and my plug-in is already processing in larger block sizes?

A: Depending on how your custom buffering is implemented, you may or may not see an improvement in performance. However, we still strongly encourage you to take advantage of the variable RTAS quantum. This will help keep the plug-in up-to-date with the latest changes in this area. In the case that you need to also support custom buffering just for older versions of Pro Tools, you can use the Fic gestalt mentioned in question 1.

RTAS Stem Formats on Pro Tools TDM

Previously, it was required that RTAS plug-ins have symmetrical (N x N) input/output stem formats on TDM tracks in Pro Tools. Beginning in Pro Tools 7, this is no longer the case. RTAS plug-ins are now capable of changing the width of a TDM track, i.e. Mono Input -> Stereo Output, etc. In other words, a mono TDM track can have an RTAS plug-in with asymmetrical (N x M) stem formats. In addition, RTAS plug-ins may now be inserted anywhere in the insert signal path, either before or after any TDM plug-ins.

RTAS on Aux Inputs/Master Faders

In Pro Tools 7.0 and later, RTAS audio processing and instrument plug-ins are able to be inserted on Aux Input and Master Fader tracks. This has some important implications for developers:

Null Connections Fix

RTAS instrument plug-ins should be able to operate with no input assigned. For RTAS plug-ins not using the true-RTAS implementation (i.e. `RenderAudio` in `CEffectProcessRTAS`) but using the gestalt `pluginGestalt_WantsCallbackOnNullConnections`, this will require updating to a Pro Tools 7.0 or later plug-in SDK and (potentially) making minor changes in the plug-in code.

In versions of Pro Tools and the plug-in SDK prior to 7.0, a plug-in's input `DAEConnection` structures contained the number of samples to process in its `mNumSamplesInBuf` field, and the output `DAEConnection` structures contained a zero in this field. However, when no input was connected to the plug-in, no input `DAEConnection` structures existed, so the plug-in had no good way of knowing how many samples to process.

This was fixed in Pro Tools 7.0: if no input `DAEConnection` structures are present, then the output `DAEConnection` structures will contain the number of samples to process in its `mNumSamplesInBuf` field.

RTAS instrument plug-ins built with the Pro Tools 7.0 or later plug-in SDK will need to first check for input `DAEConnections`, and then check for output `DAEConnections` if no inputs are connected. Previously, Digidesign RTAS instrument plug-ins would simply fail to process audio when no inputs were connected - this will be fixed in the Pro Tools 7.0 versions of these plug-ins. Third-party plug-ins with this problem should be similarly fixed.

NOTE: This means that if an RTAS plug-in is using ProcessAudio() instead of RenderAudio() as its RTAS callback function, it will have to be updated to the 7.0 or later plug-in SDK and fixed as described above. Without doing this, the plug-in will not behave properly in versions greater than Pro Tools 7.0. When placed on an aux track or instrument track, it will not produce audio unless an input is connected.

RTAS on Instrument Tracks and DirectMIDI

Beginning in Pro Tools 7.0, when an RTAS instrument plug-in is instantiated on an instrument track, the output is auto-assigned by Pro Tools. However, if your plug-in doesn't use DirectMIDI nodes, the output will not be auto-assigned. Instead, the user will have to manually choose the output on the track. This

problem only occurs on Windows, as it is only possible to use DirectMIDI nodes on Mac. For this reason, in addition to the fact DirectMIDI is the supported cross-platform MIDI protocol of Pro Tools, it is strongly suggested all plug-ins using legacy OMS MIDI nodes port to DirectMIDI.

Chapter 10: DirectMidi

DirectMidi is Digidesign's specific protocol for communication of MIDI and other timing critical information from Pro Tools to plug-ins. DirectMidi is not intended to supplement Apple's CoreMIDI services on OSX. Rather, it is a cross platform solution to tightly integrate Pro Tools, DAE, and plug-ins. It was created to handle the MIDI capabilities of plug-ins, and it will continue to enable feature-rich MIDI plug-ins to take advantage of the expanding MIDI functionality in Pro Tools.

The MIDI standard is maintained by the MIDI Manufacturers Association (MMA). To learn more about the MMA, visit their website at http://www.midi.org. For an online version of the MIDI Specification, please visit http://www.borg.com/~jglatt/tech/midispec.htm. Note that Developer Services provides these links as an external resource only. The links and the information contained therein are not associated with nor guaranteed by Developer Services in any way.

This chapter assumes basic knowledge of MIDI and its message types. The auto-generated SDK documentation from the `DirectMidi.h` header and in the Effect Layer MIDI headers supplement the information here and go into greater detail on the specific APIs.

Architecture

As a component of Pro Tools, DirectMidi facilitates communication between MIDI plug-ins (virtual instruments, synthesizers, sampler plug-ins, etc.), MIDI devices (hardware controllers, keyboards, rack synthesizers, etc.), and Pro Tools MIDI elements. MIDI plug-ins connect to the DirectMidi Plug-In Interface to receive MIDI data from and send MIDI to other MIDI components in Pro Tools.

In the land of DirectMidi, a `DirectMidiPacket` is the most basic currency by which MIDI data is exchanged. MIDI data is sent to the plug-in as a stream of `DirectMidiPacket` objects. Each `DirectMidiPacket` represents a single MIDI message and includes a time stamp, the message stored in a 4-byte array, and the length of the message. MIDI messages range from 1- to 4- bytes in length.

Local MIDI Input Nodes

Incoming MIDI data is streamed to the plug-in via local MIDI nodes. These are essentially entry points by which MIDI can reach the plug-in. The plug-in is responsible for creating these local input nodes. Upon creation, these nodes will show up in Pro Tools as assignable MIDI outputs. The user can then route the MIDI printed on a track or sent from an external device to these nodes via the Pro Tools output menu.

There are a few different "styles" of local nodes a plug-in can create:

Direct Delivery Direct Delivery input nodes receive MIDI messages one at a time through a callback the plug-in defines. DAE calls this function whenever there is a MIDI message to deliver to the plug-in.

Buffered Delivery Buffered input nodes receive MIDI by accessing buffers filled with MIDI messages. They were created especially for host-based plug-ins. RTAS plug-ins can get a buffer of MIDI data that corresponds to the current audio buffer being computed. Then, the plug-in can step through this MIDI

buffer like a “script” to respond to MIDI events within the audio callback. Buffered MIDI is very useful for plug-ins performing sequencing, sampling, and synthesis.

There are two types of buffered input nodes: RTAS- and other- buffered nodes. RTAS-buffered input nodes are automatically filled with MIDI data alongside the audio buffers every audio processing callback. Other-buffered input nodes are filled with MIDI data through calls made by the plug-in. With other-buffered nodes, the plug-in can decide when to fill its MIDI buffers and from where on the timeline to get the MIDI data.

Choosing a Node Type Deciding on which type of node to use depends largely on the needs of the plug-in. If the plug-in is TDM, it will typically implement Direct Delivery nodes. RTAS plug-ins will usually use RTAS-buffered nodes, unless the plug-in has some special needs for when and where it receives MIDI data from. In that case, the plug-in can use other-buffered nodes. All types of nodes can be implemented to handle MIDI with sample accuracy. Keep in mind that plug-ins are not at all constrained to only use the nodes in these ways.

Global MIDI Input Nodes

Additionally, a plug-in can receive streaming global data like Click, MIDI Time Code, and MIDI Beat Clock messages through a special kind of node called a global input node.

Global Input Nodes

Global MIDI nodes are like local MIDI nodes, except they do not show up as assignable outputs in Pro Tools. Instead the MIDI data is automatically routed to the plug-in, without the user making any connections. There are three different types of global nodes, corresponding to the different kinds of local MIDI nodes:

Direct Delivery Global Node A Direct Delivery node that receives global MIDI messages. Like any Direct Delivery node, MIDI messages come one at a time to a plug-in-defined callback.

RTAS Shared Buffer Global Node An RTAS-buffered node that receives MIDI data into a global buffer that is shared among all plug-in instances in a session. There may be both explicitly requested data and data not requested by the plug-in in the buffer. For example, if one plug-in requests MTC and another plug-in requests Click, all plug-ins connected to this global node will get both MTC and Click messages in the shared buffer. Like all RTAS-buffered nodes, this global node is automatically filled with MIDI data alongside the audio buffers every audio processing callback.

Other-Buffered Global Node An other-buffered node that receives MIDI data into a global buffer. Unlike the RTAS Shared Buffer Global Node, this buffer is local to each plug-in instance. Data requested by other plug-in instances will not show up here. The buffer is updated by the plug-in in the same manner as all other-buffered nodes: the plug-in gets to specify when the buffers are filled and from over what sample interval it gets the MIDI messages.

Global Messages

The types and formats of global messages are as follows:

Click The click messages are special 2-byte messages encoded as such:

- Accented click: “Note On” pitch 0 (0x90 0x00)
- Unaccented click: “Note On” pitch 1 (0x90 0x01)

“Note Off” messages are never sent. Click messages are not in the MIDI Specification; we created them as a simple way to encode click events.

MIDI Beat Clock (MBC) This includes Song Position Pointer, Start/Stop/Continue, and Midi Clock (F8), as defined in the MIDI Specification. A plug-in can also listen for MBC on a local node. See the "Miscellaneous" section for more details on receiving MBC via a local node.

MIDI Time Code (MTC) The Standard MIDI Time Code format.

Local MIDI Output Nodes

Starting in Pro Tools 7.2, plug-ins can create MIDI output nodes that will appear as MIDI streams routeable to MIDI track inputs and elsewhere. Most of the discussion of this section applies equally to both input and output nodes. Although, currently only local RTAS Buffered Output nodes are available. So any info concerning RTAS buffered nodes in of particular interest to developers wanting to implement MIDI output in a plug-in. Disscussions of Direct Delivery or Other buffered nodes will not apply to output nodes.

RTAS Buffered Output Nodes These nodes send MIDI by filling buffers with MIDI messages. RTAS plug-ins can generate a buffer of MIDI data that corresponds to the current audio buffer being computed. This is done by constructing DirectMIDIPacket objects and placing them in the output buffer provided by DirectMIDI. Details about implementing MIDI outputs in a plug-in are provided later in this section.

Caveats regarding MIDI output nodes:

- Currently if you chain together the MIDI input and output of a bunch of MIDI plug-ins, there will be one RTAS buffer's worth of latency for every plug-in after the first. This is a limitation of our existing system.
- MIDI Outputs Only Support Variable-Quanta RTAS Callbacks
- RTAS Engine Synchronization The delivery of MIDI to Pro Tools is synchronized to the audio callbacks much like the delivery of MIDI from Pro Tools is.
- Timestamping Full timestamping of MIDI messages is supported. For MIDI outputs, timestamps are to be given to Pro Tools relative to the first sample of the RTAS processing buffer.
- Sysex Support The delivery of variable length sysex messages are supported. Unlike with MIDI inputs, there are no buffer size limitations here.
- Input Enable Menu Plug-in MIDI outputs will not show up in this menu. The menu is really a feature for h/w MIDI devices.
- All Input Selector On a MIDI track, you select this to receive MIDI from all sources in the system. Again, this is a feature for h/w devices, so plug-ins MIDI outputs are excluded here.
- Non-Realtime MIDI There is a feature that allows any keyboard controller on the system to modify non-realtime MIDI events in Pro Tools. This doesn't make sense for plug-ins, so plug-in MIDI does not modify these events.

TDM Deck Running Time

At the heart of DirectMidi's timing-critical synchronization capabilities is the TDM deck running time counter. Under the hood, DirectMidi uses this to time stamp and synchronize the delivery of MIDI messages. The running time counter is a sample-accurate counter that is maintained by the TDM deck. It starts at zero when the deck is first acquired and keeps incrementing until the deck is de-acquired. For Direct Delivery nodes, a message type `eDeacquireMessage` is sent to the plug-in-defined callback every time this happens. The counter will wrap back around to zero once the deck is reacquired, or when it has incremented up to the maximum value it can hold in its given bit count (i. e. `0xfffff....ffff`). The latter depends on the "rendition" of the running time counter.

Running Time Renditions Natively, the TDM deck running time counter is 24-bits. However, the plug-in might see it as a 19-bit counter or a 32-bit counter depending on where it grabs the counter from. We

will call these different “renditions” of the counter. Although there are different renditions, they all reflect the same counter and are synchronized to the same clock.

To account for the inconsistency created by all the different renditions, you should mask the running time counter to the lowest number of bits the plug-in will see it as. This will be 19-bits. One nice thing about masking is that the wraparound logic of the counter is preserved. The counter also remains synchronized with its other higher bit-count renditions.

Accessing Running Time The TDM deck’s running time counter is made available to a plug-in in two ways:

■ For RTAS plug-ins, the running time is included as part of the audio processing callback’s param block. The newer style true-RTAS plug-in implementation makes global RTAS Engine information like this accessible to the plug-in. Running time is stored in a member variable of the `mRTGlobals` object called `mRunningTime`. `mRunningTime` is a 32-bit rendition of the running time counter and increments 32 samples at a time every audio processing callback. `mRTGlobals` is a member of the `CEffectProcessRTAS` and gets set before every call to `RenderAudio()`. Here’s how you would access `mRunningTime` in your plug-in:

```
void MyTrueRTASProcess::RenderAudio(float** inputs, float** outputs, long
frames)
{
       int runningTime = this->mRTGlobals->mRunningTime;
                              .
                              .
                              .
}
```

■ For TDM plug-ins, the running time is broadcast on the TDM bus. The running time here is a 19-bit rendition and increments one sample at a time. The plug-in may request this by adding the gestalt `pluginGestalt_WantsTDMRunningTime`. An example of how to listen for running time on the bus is in the SampleClick MuSh plug-in.

Time Stamping

Each incoming MIDI message can be time stamped with a 19-bit sample location it is due to occur at. This allows for the early delivery and processing of MIDI messages, thereby keeping latency as low as possible. This also enables MIDI messages to be handled with sample accuracy.

Time stamping can be enabled by setting the `advanceScheduleTime` parameter to a nonzero value. If it is not enabled, all messages are time stamped with a 0. Enabling can be done when passing the parameter in at the creation of a node, or through a specific API call. The “Implementation” section will describe in greater detail how to set this value.

The meaning of a time stamp differs depending on the type of MIDI node the message is being streamed into:

Direct Delivery Time Stamps For Direct Delivery nodes, a time stamp represents the sample offset relative to the TDM deck's current running time.

These time stamps may be misinterpreted when the running time counter wraps around. If unaccounted for, this may cause stuck notes. After the counter is reset to 0, the plug-in may have some MIDI messages still waiting to be processed. These MIDI messages would then have a time stamp relative to the counter before it was reset. The plug-in might look at such a time stamp and not handle the corresponding message until the next time the counter reaches that sample number. A plug-in should implement some logic to gracefully handle these cases. For example, flush any MIDI messages the plug-

in is holding onto whenever the running time wraps around, and immediately handle any late messages that come in thereafter.

Since time stamps are always 19-bits, you'll want to compare them to the 19-bit rendition of the TDM deck running time.

Buffered Delivery Time Stamps For buffered nodes, a time stamp represents the nth sample since the beginning of the current buffer. For RTAS-buffered nodes the beginning of the buffer is the start of the audio buffer being processed by the plug-in's audio callback (i.e. RenderAudio()). For other-buffered nodes, the beginning of the buffer is the running time position `runningTime` passed in when the plug-in calls `GetMIDI()` to fill its MIDI buffers (more on this later). This is a convenient representation because it corresponds directly to a host plug-in's audio processing. Also, all the wraparound logic required for Direct Delivery time stamps is implemented behind the scenes by DirectMidi.

Transport Information

There are several DirectMidi APIs that query Pro Tools for transport information. Some of these APIs give timing critical information like tick position, loop position, and playback status. Other APIs can query the transport for the current tempo and meter wherever the transport is playing through or stopped at. For more details and caveats on calls for transport information, please refer to the auto-generated SDK documentation.

When the transport is playing, all transport information is updated once per Hardware Buffer Interrupt. Hardware Buffer Interrupts directly correlate with the H/W Buffer Size setting in Pro Tools. When the transport is not playing, the transport information is updated every time the plug-in queries for it.

Tick Position Tick position is a running counter based off the transport and tempo of a session. This is NOT the same as the TDM deck running time counter. There are 960,000 ticks per quarter note in Pro Tools. Tick zero corresponds to bar 1: beat 1 of the Pro Tools session.

Versioning

DirectMidi is versioned as new features and APIs are added. Recent versions of DirectMidi may not be available on earlier versions of Pro Tools. For specifics on what versions of Pro Tools correspond to what versions of DirectMidi, see the auto-generated documentation.

Effect Layer Implementation

Implementing DirectMidi is fairly straightforward when using the Effect Layer. DirectMidi version checking is automatic and all the different nodes are separate class objects. To add MIDI to your plug-in, begin by adding code as such:

```
//Connect to the DirectMidi Plug-In Interface
result = MIDILogIn();


//Create the node object(s)
yourNode1 = new CEffectMIDIDirectDeliveryNode(&mMIDIWorld, 2048,
yourDirectDeliveryCallback, reinterpret_cast<UInt32>(this), eLocalNode,
const_cast<char *>(nodeName));

yourNode2 = new CEffectMIDIRTASBufferedNode(&mMIDIWorld, eGlobalNode);
```

```
yourNode3 = new CEffectMIDIOtherBufferedNode(&mMIDIWorld, kMaxRTASHwBufferSize,
eLocalNode,  nodeName, yourOwnBuffer);

// MIDI Output: Set up output node for MIDI thru
yourOutNode = new CEffectMIDIRTASBufferedOutputNode( &mMIDIWorld,
nodeName, kMaxRTASHwBufferSize);


//Initialize the node object(s)
yourNode1->Initialize(<channelMask>, false);
yourNode2->Initialize(0x0FFFF, false);
yourNode3->Initialize(0x0001, true);
yourOutNode->Initialize( 0x0003 );    // Channel 1 and channel 2
```

Implementing Effect Layer Direct Delivery

For Direct Delivery nodes, a callback into the plug-in must also be implemented. In the above sample code, the callback is passed in as the argument `yourDirectDeliveryCallback`. The callback is where a Direct Delivery node receives MIDI, one message at a time. Here is its type definition:

```
typedef void (*DirectDeliveryStaticCallback)(DirectMidiCallbackType type,
DirectMidiPacket* iMessage, Cmn_UInt32 nodeRefCon);
```

Implementing Effect Layer Buffered Delivery

Buffered input nodes obtain MIDI via the following calls:

```
//Obtain a pointer to the buffer of MIDI
yourMIDIBuffer = yourLocalNode->GetBufferPtr();

//Obtain the size of that buffer
yourMIDIBufferSize = yourLocalNode->GetBufferSize();
```

Note in the initial sample code that a custom buffer `yourOwnBuffer` can also be optionally specified when a buffered node is constructed. A plug-in may want to use such a buffer so that its MIDI data is accessible outside of DirectMidi, or resides in shared memory.

Additionally, other-buffered nodes must make the following call to have their MIDI buffers filled:

```
result = yourNode3->FillMIDIBuffer(mRTGlobals->mRunningTime, mRTGlobals-
>mHWBufferSizeInSamples);
```

This call fills the MIDI buffer with MIDI data over a requested range.

To fill buffered output nodes with data, use:

```
anRTASBufferedOutputNode->PushMIDIPacket(DirectMidiPacket *inPacket);
```

or,

```
anRTASBufferedOutputNode->CopyMIDIPacketToIndex( DirectMidiPacket
*inPacket, Cmn_UInt32 inIndex);
```

Error Checking

If anything goes wrong in an Effect Layer MIDI call, that function will return an appropriate error. Issues like DirectMidi version mismatches are handled this way. If the Effect Layer encounters an API that is not supported in the running version of Pro Tools, it will return an error.

Please check errors and handle them accordingly whenever possible.

Obtaining Transport Information

The `CEffectMIDITransport` class contains several API's for obtaining transport information. Simply create an object of this class in the plug-in and make the function calls for pertinent information like so:

```
//Create the transport object
yourTransportObject = new CEffectMIDITransport(&mMIDIWorld);

//Example transport object calls
result = yourTransportObject->GetCurrentTempo(&tempo);
result = yourTransportObject->GetCurrentMeter(&numerator, &denominator);
result = yourTransportObject->IsTransportPlaying(&playing);
```

Clean-Up

Once a plug-in has finished using a node, make sure to call `yourNode->DeleteNode()` so that all the necessary clean-up is executed. Also, don't forget to destroy the node object before it's all said and done.

A plug-in should also call `MIDILogOut()` in its destructor. `MIDILogOut()` deletes and destroys every node, and also frees up the plug-in's entire MIDI environment. Not calling `MIDILogOut()` may lead to undesirable remnants in Pro Tools, like memory leaks.

Examples

The sample plug-ins contain various examples of how to implement MIDI using the Effect Layer. Here is a chart mapping out which plug-ins demonstrate which MIDI feature.

Sample Plug-In	Direct Delivery Local Input Node	Direct Delivery Global Input Node	RTAS-Buffered Local Input Node	RTAS-Buffered Global Input Node	Other-Buffered Local Input Node	Other-Buffered Global Input Node	RTAS Buffered Local Output Node	Tranport Class	Local Input Node MIDI Beat Clock
SampleClick RTAS			X	X				X	
SampleClick MuSh	X	X						X	
Microbe	X								
MicrobeSampler					X	X	X		X

For a reference on the Effect Layer MIDI API, please refer to the auto-generated SDK documentation. There every argument and function is documented in greater detail.

Non-Effect Layer Implementation

Here we will help you get started using the DirectMidi Plug-In Interface without the Effect Layer. We will cover the basic implementation needed to have MIDI data streamed into a plug-in. We will not take the time here to detail every parameter and function as these are documented in the auto-generated SDK documentation.

Connecting to the Interface

A plug-in instance needs a pointer to the DirectMidi Plug-in Interface with which to call all the DirectMidi APIs. To obtain this pointer, the instance needs to register with DirectMidi by calling the following:

```
DirectMidi_RegisterClient( Cmn_UInt32 versionRequested, CDSPProcess*
thisPlugInProcess, Cmn_UInt32 clientRefCon, void** myDirectMidiPtr)
```

You'll want to request a version of DirectMidi that supports all the APIs you'll call. In the case that the requested version is unavailable, the plug-in needs to either disable its unsupported capabilities, or fail gracefully.

Once a plug-in instance registers with DirectMidi, a pointer to the interface will be passed back in `myDirectMIDIPtr`. This pointer is given to that plug-in instance, and that instance alone, to create nodes, query for transport information, etc.

When the instance is done using the interface, it should call `DirectMidi_FreeClient(void* instance)` to un-register and clean up the memory space that was being used.

Creating MIDI Input Nodes and Receiving MIDI Data

To begin receiving MIDI data, the plug-in first needs to create at least one MIDI input node. All local input nodes need to be created with an index `nodeIndex` and a name `nodeName`. The `nodeIndex` is what DirectMidi uses to reference the node. The `nodeName` will appear as the plug-in's assignable MIDI output in Pro Tools. The open, assignable channels of that output are defined by the `channelMask`. Users can route MIDI into any of these defined channels.

Pro Tools renumbers the channel of incoming MIDI messages so that it reflects the channel number of the plug-in, not the channel number of the MIDI source.

None of these parameters apply when creating global nodes, since they have no routable connections and exist one per session or plug-in instance. Instead there is a `globalBitmask` argument, where the plug-in specifies the global data it wants to receive.

Now let's take a look at how each type of node is specifically created, and then subsequently used.

Implementing Direct Delivery Input Nodes

To create a Direct Delivery input node, call the following:

```
myDirectMidiPtr->CreateDirectDeliveryMidiNode (Cmn_UInt32 nodeIndex, char
*nodeName, Cmn_UInt32 channelMask, Cmn_UInt32 advanceScheduleTime,
DirectMIDICallback myPluginCallback)
```

To create a global node , call this:

```
myDirectMidiPtr->CreateDirectDeliveryGlobalNode(Cmn_UInt32 globalBitmask,
Cmn_UInt32 advanceScheduleTime, DirectMIDICallback myPluginCallback)
```

Unique to Direct Delivery implementation, you will need to pass into these calls a static callback function `DirectMIDICallback myPluginCallback`. This function is defined by the plug-in and takes on the following form:

```
void MyProcess::myPluginCallback(DirectMidiCallbackType type, DirectMidiPacket*
iMessage, Cmn_UInt32 clientRefCon, Cmn_UInt32 nodeIndex)
```

MIDI data streams into the plug-in one message at a time as DAE calls this function. `clientRefCon` is passed back to the plug-in as well. Typically, this is a pointer to the particular plug-in process at hand. You can use this to call a non-static callback and to access an instance's member variables. This argument is originally passed into DirectMidi by the plug-in during registration.

To delete a node, call:

```
myDirectMidiPtr->DeleteDirectDeliveryMidiNode(Cmn_UInt32 nodeIndex).
```

Implementing Buffered Delivery Input Nodes

Buffered input nodes are created and implemented differently, depending on the type of buffered node.

RTAS-Buffered Input Nodes For local nodes, these are created by this call:

```
myDirectMidiPtr->CreateRTASBufferedMidiNode(Cmn_UInt32 nodeIndex, char*
nodeName, Cmn_UInt32 channelMask, void* optionalBufferToUse = NULL)
```

To connect to the RTAS shared buffer global node, make this call:

```
myDirectMidiPtr->ConnectToRTASSharedBufferedGlobalNode(Cmn_UInt32 globalBitmask)
```

That's all the implementation needed to stream MIDI data into your buffers for RTAS-buffered nodes! The buffers are automatically updated right before every `ProcessAudio/RenderAudio` callback. They are updated with all the MIDI data in the sample range of each callback. The duration of that range directly correlates to the size of the RTAS processing quanta.

Other-Buffered Input Nodes These are created by calling:

```
myDirectMidiPtr->CreateOtherBufferedMidiNode(Cmn_UInt32 nodeIndex, char*
nodeName, Cmn_UInt32 channelMask, Cmn_UInt32 advanceScheduleTime, void*
optionalBufferToUse = NULL)
```

for local nodes and:

```
myDirectMidiPtr->CreateOtherBufferedGlobalNode(Cmn_UInt32 globalBitmask,
Cmn_UInt32 advanceScheduleTime, void* optionalBufferToUse = NULL)
```

for a global node.

For these nodes, a plug-in needs to fill the MIDI buffers by calling:

```
myDirectMidiPtr->GetMIDI(Cmn_UInt32 runningTime, Cmn_UInt32 bufferLength)
```

The buffers will be filled with all incoming MIDI messages scheduled to occur at a time that is equal to or greater than `runningTime` and less than `(runningTime+bufferLength)`.

The `advanceScheduleTime` parameter tells the DirectMidi how early messages need to be ready for a call to `GetMIDI()`. This needs be set to far enough in advance so that all the pertinent MIDI messages will ready by the time the plug-in actually calls `GetMIDI()`. At any time, a plug-in can call `myDirectMidiPtr->SetAdvancedScheduleTime()` to change this.

To delete an RTAS- or other- buffered node, call:

```
myDirectMidiPtr->DeleteBufferedMidiNode(Cmn_UInt32 nodeIndex).
```

Accessing Input Buffers RTAS- and other- buffered nodes share the same space for their buffers. The buffers can be accessed via an array of node pointers `mNodeArray`. This array is a member of the struct `DirectMidiParamBlock`. The specific instance of this struct that DirectMidi manages is pointed to by the `directMidiParamBlkPtr`. A node's `nodeIndex` is its index into the `mNodeArray`. Here's an example showing how to the access the buffer of a node created at index 5 and print out the MIDI messages in that buffer:

```
myMidiBufSize =
myDirectMidiPtr->directMidiParamBlkPtr->mNodeArray[5]->mBufferSize;

myMidiBufferPtr =
myDirectMidiPtr->directMidiParamBlkPtr->mNodeArray[5]->mBuffer;

for (int i=0; i<myMidiBufSize; i++)
{
       for (int j = 0; j<myMidiBufferPtr->mLength; j++)
       {
              printf("%x", myMidiBufferPtr->mData[j]);
       }
       myMidiBufferPtr++;
}
```

Custom Input Buffers Optionally, plug-ins can pass in their own buffers to be filled by specifying the `optionalBufferToUse` parameter when the node is created.

Receiving MIDI Beat Clock in Local Input Nodes

In addition to being sent to global input nodes, MBC can be routed to local input nodes as well. The plug-in can enable this by calling:

```
myDirectMidiPtr->SetDirectDeliveryMidiNode_ReceivesMBC(Cmn_UInt32 iNodeIndex,
bool iEnable)
```

for Direct Delivery nodes and:

```
myDirectMidiPtr->SetBufferedMidiNode_ReceivesMBC(Cmn_UInt32 iNodeIndex, bool
iEnable)
```

for buffered nodes.

When nodes are created, this feature is enabled by default.

If enabled, the plug-in node will show up in the MIDI Beat Clock menu in Pro Tools. The user can then route MIDI Beat Clock data to the plug-in node if they wish. Plug-ins already receiving MBC through a global node may want to disable this feature.

Creating MIDI Output Nodes and Generating MIDI Data

To allow a plug-in to send MIDI data to Pro Tools, the plug-in first needs to create at least one MIDI output node. All local output nodes need to be created with an index `nodeIndex` and a name `nodeName`. The `nodeIndex` is what DirectMidi uses to reference the node. The `nodeName` will appear as the plug-in's assignable MIDI input in Pro Tools. The open, assignable channels of that output are

defined by the `channelMask`. Users can route MIDI from any of these defined channels to any assignable MIDI input

Implementing MIDI output is analogous to implementing MIDI input. Currently only local RTAS Buffered Output nodes are available so only they will be discussed here. All previous discussion of connecting to the Direct MIDI interface are the same as for input nodes.

Implementing RTAS Buffered Output Nodes

RTAS-Buffered Output Nodes For local nodes, these are created by this call:

```
myMIDIInterface->CreateRTASBufferedMidiOutputNode(Cmn_UInt32 nodeIndex, char*
nodeName, Cmn_UInt16 channelMask, Cmn_UInt32 maxBufferSize)
```

And to place data in an output buffer:

```
DirectMidiNode* outputNode =
       mMIDIInterface->directMidiOutputParamBlkPtr->mNodeArray[nodeIndex];

//copy a packet of midi to the output buffer
DirectMidiPacket packet;
MyPacketFillerFunc( packet );
std::memcpy(outputNode->mBuffer, &packet, sizeof(DirectMidiPacket));
outputNode->mBufferSize++;
```

That's all the implementation needed to send MIDI data from your plug-in for RTAS-buffered output nodes.

Miscellaneous

This section contains information about other miscellaneous MIDI items.

Threads and Critical Sections for MIDI

For the most part, DirectMidi does not introduce any threading complications for plug-ins. Any calls to the DirectMidi interface, creation/deletion of nodes, and MIDI buffer accesses are thread safe for the plug-in.

You do have to be careful with Direct Delivery nodes when you are in the thread that calls the plug-in-defined callback. The callback is directly called by one of the Pro Tools MIDI threads and bypasses the Dispatcher. Therefore, it is not thread safe when it reaches the plug-in. Special care should be taken so that this thread doesn't conflict with any others. Any calls or operations the plug-in makes in this thread need to be made thread safe by the developer.

One specific danger area pertaining to this is host port access. To prevent random crashes while accessing the host port from within the callback, it is essential to wrap host port reads and writes in a Critical Section. This ensures that another thread calling into the plug-in does not access the host port while you are attempting to access it.

Tech Note #13, available on the developer website, discusses Critical Sections in general in more detail.

Sample-Accurate MIDI Output for Direct Delivery Nodes

The following describes a mechanism built-in to DirectMidi to allow for sample-accurate Direct Delivery MIDI output.

Plug-ins can simply process Direct Delivery messages as they arrive, but messages would then be executed with inaccurate timing. The performance of this implementation would depend completely on the arrival time of the MIDI messages. In a real-time system like Pro Tools, that arrival is contingent on a variety of unpredictable factors.

A better implementation is to utilize the early, time stamped delivery of MIDI messages. A plug-in can buffer up the messages beforehand and output them when they are time stamped to occur. By comparing the time stamps of stored MIDI messages with the TDM deck's running time counter, a plug-in can determine when to play a message. Such a system would yield sample-accurate MIDI processing.

To program this, the plug-in instance needs to enable time stamping and tell DirectMidi to deliver MIDI messages earlier than their scheduled time-of-occurrence. The `advanceScheduleTime` parameter tells DirectMidi how much earlier than the time stamp a MIDI message should arrive. This parameter is set when a node is created. DirectMidi will then do its best to deliver messages at least as early as the plug-in specifies. It is not perfect though. Plug-ins still need to be robust enough to handle a couple of nondeterministic delivery scenarios. For one, messages may arrive even earlier than specified. Second, the system experiences jitter, and a message may occasionally arrive past its time stamp.

Despite this caveat with DirectMidi's advance delivery, the resulting implementation is still an overall improvement. The SampleClick MuSh example plug-in implements this technique in the SDK.

System Exclusive MIDI Messages

Like other MIDI messages, System Exclusive (SysEx) messages get sent to the plug-in via `DirectMidiPacket` objects. What's unique about SysEx messages though is that they can be greater than 4-bytes. Therefore, while other MIDI messages will only take up one `DirectMidiPacket`, SysEx messages can take up multiple `DirectMidiPacket` objects. DirectMidi will send SysEx messages as a stream of `DirectMidiPacket` objects. In accordance with the MIDI Specification, `0xF0` will indicate the beginning of a SysEx message and `0xF7` will indicate the end. For buffered nodes using custom buffers, SysEx messages greater than 2048 bytes cannot be delivered to the plug-in. SysEx messages delivered to Direct Delivery nodes have no size limit.

Part V: Miscellaneous

Chapter 11: General Topics

Plug-in IDs and Registering Your Plug-In With Digidesign

Plug-in ID's are an important part of the Digidesign Plug-in Architecture. In your plug-in code, you must provide a Manufacturer ID, Product ID, and Type ID. These ID's are concatenated to create a 96-bit ID to uniquely identify a plug-in to DAE and the DAE host program. These ID's are saved with persistent data – like the session file for Pro Tools – so that the host program can restore your plug-in with the user's previous settings and automation data associated with it. (See the Forwards and Backwards Compatibility section below for additional information.)

All three ID's are of the data type OSType, which is a four character code. All ID's are declared at the group level so that when DAE is loading, it can gather the list of all plug-in types and their properties (like the category, the number of inputs and outputs, etc.). In general, normal practice is that there will be one Manufacturer ID for all of your plug-ins, there will be one Product ID to identify a plug-in binary, and there will be as many Type ID's as there are process types for a particular product. Note that it is possible to have more than one Product ID in a single binary. For example, the DigiRack Dynamics plug-in contains the Compressor, Limiter, Expander and Expander-Gate plug-ins, all with different Product ID's.

As an example, let's look at the Template example plug-in. The Template plug-in declares the Manufacturer ID in the group constructor by passing `'Digi'` to the `DefineManufacturerNamesAndID()` method. The Product and Type ID's are declared in `CreateEffectTypes()` in the EffectType constructors. It passes in `'tmpl'` for the Product ID and then a different Type ID for each type it creates: `'tmAS'` for AudioSuite, `'tmRT'` for Mono RTAS, `'StRT'` for Stereo RTAS, etc.

There are some important restrictions on the characters that can be used for plug-in ID's. The characters used for ID's must be printable characters that can be used in XML and that can be used in filenames on both Mac and Windows. Therefore, acceptable ID's:

1) Use ASCII characters with values between, and including, 0x20 and 0x7E
2) Do not contain any of the following characters: `/ \ : * ? " < > | ' &`

Because DAE uses the concatenation of the three ID's to uniquely identify a particular plug-in, it is important that there be no conflicts in any ID. In order to make sure that this system works across all third party plug-in developers, it is important that you register your Manufacturer ID with Digidesign.

Contact Developer Services and request that your desired Manufacturer ID be reserved for your company. We will let you know if that ID has been taken. It will then be your responsibility to make sure that there are no conflicts between your Product and Type ID's across your entire product line. By doing so, you will be able to assure that your plug-ins will not conflict with any other registered third party plug-ins.

Forwards and Backwards Compatiblity Issues

Binary Compatibity

The move to OS X created a capability break, primarily due to Carbonization. OS 9 plug-ins will not run in OS X. OS X plug-ins will not run on OS 9. To aid in development, recent versions of the plug-in SDK have multiple targets that continue to support compilation for both platforms.

Session-Stored Automation Compatiblity

During session reload automation data must be relinked to controls within a plug-in. To accomplish this, Pro Tools stores the Str31 name of a control with the associated automation data stream. If a stream cannot be rematched to a plug-in control, the automation data is lost. Therefore, perserving control names across new versions of a plug-in is critical. However, breaking automation compatiblity may be a desired consequence in certain cases. For example, a control's range or units might change in a newer version. This invalidates the automation curves between the newer and older versions.

DirectMidi and OMS Session Compatiblity

The migration from OS 9 to OS X created a change in the MIDI infrastructure of Pro Tools. Under OS 9 and OMS, typically the '`midi`' chunk structure that stores the MIDI state contains two values: a 32-bit "`nodeID`" (of type `OMSSignature`) and a 16-bit "`uniqueID`" (of type `OMSUniqueID`). These are then used in the `OMSCreateVirtualDestination()` call to generate a MIDI node for the plug-in. Additionally, this method expects an `OMSStringPtr nodeName` argument. This is a user readable string that represents the virtual node. Conventionally, this string is generated by concatentating the plug-in's name with a string version of the `nodeID` argument. This node name is stored in the session and associated with the MIDI track. When a OS 9 session is reloaded on OS X, the newer DirectMidi system reconnects a MIDI track on the basis of the MIDI node name. Therefore, it is important that your DirectMidi-enabled plug-in utilize the OMS nodeID stored in the '`midi`' chunk to generate a node name in an equivalent form to your OS 9 plug-in. Otherwise, Pro Tools will not reconnect the node.

Unfortunately, backwards compatibility of node relinking is not guaranteed. However, to insure the best compatiblity, a DirectMidi plug-in should create an OMS-like chunk, by storing the DirectMidi node ID in place of the OMS node ID and a uniqueID of zero.

Saving And Restoring Settings (Without using the Effect Layer)

An instantiated plug-in often has state information that must be saved in order to properly re-instantiate that plug-in at a later time. The `SFicPlugInChunk` data structure and "chunk" related plug-in routines are designed to standardize the storage and retrieval of such state information. To save and restore data, you need to support at least one chunk, return *true* for the Gestalt selector `supportsSaveRestore`, a and override all methods with `Chunk` in the method name.

The Effect Layer transparently handles Save/Restore functionality for all of a plug-in's controls. If additional save and restore chunks are needed, it's still possible to override the Effect Layer's chunk methods, then filter out or add chunks before calling the inherited Effect Layer methods. See the chapter ***Using the Effect Layer*** *for details on implementing save-and-restore with the Effect Layer.*

Plug-In Settings Files

Ultimately Chunks get stored in Settings files (.tfx files) which by default reside in the "DAE\Plug-In Settings" folder. Often, all plug-in settings files for any particular plug-in will be readable by all of that plug-in's process types. However, it may be desirable to only show compatible process type plug-in settings within the plug-in's librarian pop-up menu (for instance, when process types belonging to one plug-in are very different, it is not appropriate to be able to view all the process type settings from the librarian pop-up menu). By specifying a different Product ID for each of the process types, this can be accomplished. This filtering of Product IDs is performed by Pro Tools and not the Plug-In Library.

Chunk Methods

There are seven chunk related methods that must be handled for save-and-restore functionality. Here they are listed with brief descriptions of their functionality.

CompareActiveChunk() Used by Pro Tools to determine whether not to illuminate the Compare light.
DescribeChunk() Return a descriptive string for the chunk.
GetChunk() Copy the parameter information into the empty chunk structure that DAE provides.
GetChunkSize() Return the require size of your chunk so DAE can allocate enough space.
GetIndexedChunkID() Return an OSType ID for a particular index.
GetNumChunks() Tell DAE how many chunks the plug-in uses.
SetChunk() DAE is passing in a saved chunk. Initialize the plug-in with this new state.

Chunk Structure

A chunk is merely a handle to a chunk of data of arbitrary length that you set. The structure is of type SFicPlugInChunk defined as follows.

```
typedef struct SFicPlugInChunk
    {
        long fSize;                     // The size of the entire chunk (including this field).
        long fVersion;                  // The chunk's version.
        OSType fManufacturerID;         // The plug-in's manufacturer ID (assigned by Digi)
        OSType fProductID;              // The plug-in file's product ID
        OSType fPlugInID;               // The ID of a particular plug-in within the file
        OSType fChunkID;                // The ID of a particular plug-in chunk.
        Str31 fName;                    // A user defined name for this chunk.
        char fData[1];                  // The first byte of the chunk's data.
    } SFicPlugInChunk, *SFicPlugInChunkPtr;
```

Each plug-in may have several "chunks" of state data that can independently be saved and restored. A reverb plug-in, for instance, might wish to save room shape data independently from wall characteristics. Each of a plug-in's chunks is uniquely identified by an OSType called its chunkID. The reverb chunks in the example above might be 'Room' for room shape and 'Wall' for wall characteristics. The routine GetIndexedChunkID() returns a chunkID when passed an index which ranges from 1 to the value given by GetNumChunks().

Just as in our example, it is useful to know what kind of state data a chunk contains. The routine DescribeChunk() returns a short string that describes a particular chunkID. The reverb plug-in would return "Room Shape" for 'Room', and "Wall Characteristics" for 'Wall'. Pro Tools does not call this method, but other DAE applications may.

A chunk is stored as a block of data preceded by a header. The first field of the header is a 4-byte value containing the entire size of the chunk, including these 4 size bytes. The next field is a 4-byte value that specifies the chunk's version number. The following 3 fields are OSTypes that uniquely determine the type of plug-in that the chunk belongs to. These 3 OSTypes are the same as those from the SFicPlugInSpec data structure. The next field is the chunk's chunkID.

After the chunkID is a string containing the user defined name of the chunk. A user of our example reverb might have two 'Wall' chunks named "Fly Paper Wall" and "Jello". The name is followed by a variable sized array of characters containing the chunk's data. Because the size of chunk's data is

variable, the routine `GetChunkSize()` has been provided to return this information. The size returned includes the entire size of the chunk and its header including the 4 bytes for the `fSize` field itself.

The routine `GetChunk()` fills in a `SFicPlugInChunk` data structure with the data for a particular `chunkID`. All header information is filled in as well. The data block passed to the routine must be big enough to hold the chunk.

The routine `SetChunk()` restores a chunk's data from a `SFicPlugInChunk` data structure.

The routine `CompareActiveChunk()` compares a plug-in's current state with the passed-in chunk. If the chunks are equal, `isEqual` should be set to true. Otherwise, `isEqual` is set to false. Lastly, when storing chunk data, continuous control parameter values should be stored as long data types, as discussed in the section **Encoders and Small Relative Token Changes**.

Compare Light Functionality

Comparing current plug-in settings to any passed in chunk settings is done via the plug-in library method `CompareActiveChunk()`. This allows the Compare light in Pro Tools to activate if the user manually moves any control from its original saved position; or, likewise, to deactivate the compare light if the control is once again returned to the original position. `CompareActiveChunk()` also enables compare light functionality for external controllers.

The implementation of CompareActiveChunk() for continuous controls changed starting in Pro Tools 7.0. Previously, continuous controls were compared using their actual control values, which are passed from DAE to the plug-in via tokens. Unfortunately, these tokens occasionally have single-bit errors which are insignificant to the audio signal, but can incorrectly activate the compare light. Now the values are compared on the basis of their text display, i.e. the value that is displayed on the screen or control surface. This means that if you have a gain control with a step size of .3 dB, the compare light will not light up until the user has modified the control by .3 dB or more. The implementation for discrete controls has remained unchanged.

Factory Default Settings

Every plug-in has a `<factory default>` setting that is selectable by the Pro Tools user under the librarian settings menu. This setting is merely the chunk, or set of chunks, that the plug-in returns when its `GetChunk()` method is first invoked, before the user or DAE has altered its state. In other words, it's the preprogrammed plug-in parameters created by the developer.

Accessing the Resource Fork

The static routines `CProcessGroup::UseResourceFile()` and `CProcessGroup::RestoreResourceFile()` should be used to wrap every `.rsr` file access. This will not only ensure that you are saving and restoring the currently cached `.rsr` file, but prevent you from blowing away the `.rsr` file that was previously in the cache. Here is a code snippet showing typical usage.

```
// save currently cached res file
short SaveResFile = CProcessGroup::UseResourceFile();

theTable = Get1Resource('WTBL',ResourceID);

if (theTable != NULL)
{
     DetachResource(theTable);
     CProcessGroup::RestoreResourceFile(SaveResFile);
}
else
{
```

```
        OSErr anErr = ResError ();
        CProcessGroup::RestoreResourceFile(SaveResFile);

        if (anErr)
        FailOSErr (anErr);   // bail if res manager bailed.
    }
```

Cacheable Plug-Ins

To speed up launching DAE, and to reduce memory requirements, plug-ins are cached. When cached, DAE remembers all the process types within a plug-in so that the DAE application can show the availability of a plug-in without the plug-in yet being loaded into RAM. The first time DAE is launched with a new plug-in in the plug-ins folder, a file is created with the cached information. This file is called "`Installed PlugIns`" and is located in the DAE Prefs folder.

To allow caching, the plug-in Gestalt selector `pluginGestalt_IsCacheable` must return *true*. It is placed at the process group level, because the entire plug-in file must be either cacheable or not. If you are utilizing the Effect Layer, you need to add the following line to your Group constructor.

```
AddGestalt(pluginGestalt_IsCacheable);
```

All plug-ins should support this gestalt except those that possibly depend on special hardware and change their process types depending on what's installed. For example, the SampleCell plug-in adjusts its Process Types depending on how many SampleCell TDM cards it sees in the system. If this information had been cached, it would not get a chance to be updated properly. Therefore, the SampleCell plug-in returns *false* for the "Is Cacheable" gestalt. This is the rare case though — in other words, virtually all plug-ins should be cached!

Beginning in DAE 5.3, the plug-in cache is rebuilt anytime the MIX or HD card list changes from the previous load. This allows plug-ins to correctly add their Types depending on the available hardware resources.

Starting with DAE 5.3.1, placing a file named "`AlwaysRebuildCache`" in the "`DAE Prefs`" folder will force DAE to never use old cached information. Forgetting to force a cache rebuild is common (and very annoying!) development mistake that is solved by this mechanism.

Single-Binary MIX and HD Compatibility

With the release of Pro Tools 5.3 and the HD system, DAE on MacOS was converted to a shared library in contrast to its former implementation as an application. However, intelligence was added to "InitFic.lib" to ensure that plug-ins built with the 5.3 SDK were backwards compatible with older MIX systems.

There remains the issue that certain plug-in Types within the collective binary might not be supported on both platforms. Therefore, it may be necessary to switch off a Type depending on the platform detected. The Effect Layer already implements this functionality based on the information provided via the Type call `DefineSampleRateSupportForCardType()`. For those updating an older plug-in, you can selectively add your various Types within the `CProcessGroup::CreateProcessTypes()` method, using the `IsCardPresent(short cardType)` utility function found in `PlugInUtils.cpp`. Use the card type ID of `kSatchmoType` (for MIX) or `kGershwinType` (for HD).

Sample Rates

Starting with Pro Tools/DAE 5.3, higher than 44.1/48k sample rates are supported. A session can be set to either 44.1k, 48k, 88.2k, 96k, 176.4k, or 192k. The sample rate of a particular session is fixed. Therefore, a plug-in will not be requested to change sample rates mid-session. Its operating sample rate will be set and fixed at initialization. The Process level method `GetSampleRate()` allows the plug-in to determine the current sample rate. A `long` value is returned in units of Hertz.

A few sample rate utility functions are provided in `SampleRateUtils.cpp`, which is linked into the Plug-In Library. There are:

`long CourseSampleRate(long sampleRate)` This function converts a sample rate value to either `eSRUtils_48kRangeCourse`, `eSRUtils_96kRangeCourse`, or `eSRUtils_192kRangeCourse`.

`long CourseSampleRateFactor(long sampleRate)` Returns 1 for 48k, 2 for 96k, and 4 for 192k gross sample rates.

`long CourseSampleRateIndex(long sampleRate)` Returns 0 for 44.1k/48k, 1 for 88.2k/96k, and 2 for 176.4/192k sample rates.

Building & Testing Plug-Ins Without Relaunching Pro Tools

With DAE v5.1 - v5.3.x

It is possible to keep Pro Tools running while building plug-ins and not have to re-launch Pro Tools to see the changes. This has been made possible partly due to caching and will only succeed if there are no instances of a plug-in in Pro Tools. In order for this to work, create a file (any file, like a simple text file) and name it "`EnableCacheFiles`". The contents of the file are unimportant. Place it in the DAE Folder. Make sure you have an alias to your plug-in in the plug-ins folder. This has the added benefit that you don't actually have to copy the plug-in into the plug-ins folder after you build it. Remember that you can't have any instances of the plug-in in Pro Tools for this to work.

If your plug-in is cacheable, you can leave Pro Tools running and recompile your plug-in. This also includes any changes made to the resource fork, which will appear when you switch back to Pro Tools. Basically, this means that you can change your view resources or DSP code without having to quit Pro Tools. This should result in a large time savings for developers.

With DAE v6.0 or higher

Starting with v6.0, DAE will query the Pro Tools folder for a file named `DigiOptionsFile.txt`. There are two keywords with tokens that can be placed into this file: "`AlwaysRebuildCache 1`" and "`EnablePlugInHotSwap 1`". `EnablePlugInHotSwap` performs the same functionality of the `EnableCacheFiles` file, described above. `AlwaysRebuildCache` attacks the problem described below, by forcing DAE to requery all plug-ins every time Pro Tools is launched. It is probably ideal to enable both of these functions during your development. DAE will acknowledge the `DaeOpts.txt` file by producing a `DaeOpts.txt.ack.txt` file.

Please note that starting with Pro Tools 6.9, Pro Tools is a packaged application on the Mac. On Pro Tools 6.9 only, the `DaeOpts.txt` file will need to be placed inside the package, next to the actual executable. Control-click on the Pro Tools package, select "Show Package Contents" and navigate to the location of the executable. For Pro Tools version 7.0 and later, the DaeOpts.txt file should be located outside the application bundle and in the same folder as the ProTools application.

Issues with Caching

Let it be noted that there is a known problem with caching; if you change the number (or kind) of process types, they will not be recognized. In this case, you should trash the DAE Prefs folder (located within the system Preferences folder) and re-launch Pro Tools so that it can re-read the plug-in. There may be some other odd behavior associated with the caching mechanism, so if you are ever in doubt, just delete the DAE Prefs folder and re-launch Pro Tools. Also, be sure that you don't have any plug-ins instantiated when you go to "make" your project. Other than this, the caching mechanism has been a great boon to plug-in development, allowing Pro Tools to remain open while rebuilding plug-ins, as well as significantly speeding up the launch of Pro Tools for users.

Creating Plug-In Icons on Mac OS X

See Technote #14 on the developer website for information on creating icons for CFM plug-ins. Creating icons for Mach-O plug-ins is demonstrated in the Microbesampler sample plug-in in SDK version 7 or later. Please see the README in the MicrobeSamplerCustomIcon folder.

Copy Protection

Digidesign uses InterLok™ copy protection made by PACE Anti-Piracy, for its plug-ins. We very strongly encourage developers to also use InterLok™ for copy protection in an effort to increase system stability across the broad range of plug-ins. Copy protection schemes among the different manufacturers has been a source of problems in the past, and has kept plug-ins from co-existing peacefully in the DAE application. Therefore, the best way to prevent this is to use InterLok™ on your plug-in also. To find out more about InterLok™, you can visit the PACE Anti-Piracy web site at `http://www.paceap.com`.

Chapter 12: Working In the Development Environments

CodeWarrior Vs. Visual Studio Vs. Xcode

The sample plug-ins provided with this SDK have separate `WinBuild` and `MacBuild` folders that hold the components that specifically relate to the separate platforms. The following table attempts to cross-reference the functionality of these platform specific files.

CodeWarrior	Xcode	Visual Studio	Function
`.mcp`	`.xcodeproj`	`.vcproj`	The main Project file for the plug-in.
n/a	n/a	`.sln`	The Workspace (or Solution) that contains the Project, or multiple Projects.
`.sym`	`Compiled in the binary`	`.pdb`	Debugger symbol files.
`~PlugIn.rsr`	`~PlugIn.rsr`	`ThisPlugIn.rsr`	The Resource Fork information.
`Within .mcp`	`Within .xcodeproj`	`.def`	Specifies the Exported symbols of the plug-in.
`~SACD.r`	`~SACD.r`	`DSPCode.rcx`	Specifies the DSP code binaries to add to either the Resource File (on Mac), or the resource segment (on Windows) of the plug-in.
n/a	n/a	`.rc`	Specifies data to place in resource section (`.rsrc` segment of binary). This file points to `.rcx` file.

CodeWarrior Specifics

If you are creating a cross platform (Windows and Macintosh) plug-in, you should set the following setting in your CodeWarrior project settings, for both the Release and Debug targets: in Target -> Access Paths, enable the "Interpret DOS and Unix Paths" setting. This will allow you to easily implement cross-platform #include access paths.

For your release targets, you should set the Traceback Tables option to None. You'll find this option in the project settings under Code Generation -> PPC Processor.

Xcode Specifics

See the Xcode Porting Guide detailed below for all Xcode development considerations.

Windows Specific Files

`YourSpecializedByteSwappers``.cpp` Provides customized byte swapping routines to the Altura system to use on Resource Fork items, if custom resource types have been used.
`CompileFlags.h` PC specific conditional compile flags.

`PlugInName_resource.h` A small header file containing some Visual Studio boilerplate. Used by VS to ensure distinct resources receive distinct numeric IDs.

The following files are not found in the plug-in directories themselves. However, they are included additionally into the Visual Studio project and can be found in the Plug-In Library source tree.

`NullSwap.cpp` A default empty set of byte swappers that can be placed in the project, if there are no custom items in the Resource Fork `.rsr` file.
`DLLMain.cpp` Implements the exported DLL entry points for Windows plug-ins.

Visual Studio Project Settings

Under the menu "Project," the item "Settings..." invokes the "Project Settings" dialog for the workspace and its projects. The following table list the most critical compiler and linker settings to generate a functioning plug-in.

"C/C++" → "Category: Code Generation"	
Use run-time library	(Debug) Multithreaded DLL
Calling convention	__stdcall
Struct member alignment	2 Bytes

Starting with the 6.1 Plug-in SDK, you can use a calling convention other than __stdcall. However, your plug-in project and the plug-in library project must both be set to the same calling convention. Also starting with the 6.1 version, you can set your structure alignment to any setting larger than 2 bytes. However, the alignment for all projects in the workspace or solution must be the same.

ThisPlugIn.rsr

After a plug-in has been fully built on the Mac, the utility script `swap_resource_to_data.sh` should be executed on the plug-in binary. The resulting file is the Resource Fork data transferred to the Data Fork; the Data Fork executable data is lost. This file for the sample plug-ins is renamed to `ThisPlugIn.rsr`, and transferred to the Visual Studio project folder. It is important to run `swap_resource_to_data.sh` on the built plug-in to insure that any legacy (.r) page table resources and DSP resources are also included, since they are placed into the Resource Fork at compile time.

Since Mach-O .dpm project bundles store the resource data in the Data Fork of the bundled .rsrc file, it is only necessary to rename this file to ThisPlugIn.rsr before transferring to the Visual Studio project folder.

Built Plug-Ins

`YourPlugIn``.dpm` - the compiled DLL plug-in.
`YourPlugIn``.dpm.rsr` - The `ThisPlugIn.rsr` file is renamed and placed along with the `.dpm` file.

Runtime Type Information (RTTI)

In any development environment, it is necessary to enable RTTI in order to compile the plug-in library. Various functions in the plug-in library make use of a C++ dynamic cast, of the form `dynamic_cast<targetClass>(aSourceClassObject)`. This call requires runtime type information for the classes involved, so be sure to set this compiler option if your project does not set it by default.

Debugging

DAE/DSI Error Codes

All DAE error codes are easily recognized by their 4 digit length. They are defined and commented in the file `FicErrors.h`, which can be found in the SDK folder `/AlturaPorts/Fic/Interfaces/DAEClient/`. You will probably refer to this often during your development process! Additionally, when doing TDM development, you might encounter single or double digit error codes. These typically originate from DSI, and can be found in the file `/AlturaPorts/SADriver/Interfaces/SADriver.h`.

Sending Debug Messages to the Console

It is possible to create debug messages in a plug-in for viewing during execution. The method depends on the platform.

On OS X on Macintosh, `cout` and `printf` messages can be used. The output of the stream can be viewed in Apple's Console application, found in /Applications/Utilities.

On Windows, the Win32 API, `OutputDebugStr()`, can be used. To use this API, the source file in which it is used must include windows.h. If your plug-in uses the Altura Mac2Win translation library, i.e. includes Mac2Win.h, then please read Tech Note #8 on how to include windows.h without conflicting with the Altura definitions.

Debugging Tips

The first step to debugging a plug-in is to ensure that Pro Tools isn't working from cached information about the plug-in. To force Pro Tools to freshly query and cache all the plug-ins, delete the contents of the "`DAE Prefs`" folder before launching. Starting with DAE 5.3.2, placing a file named "`AlwaysRebuildCache`" in the "`DAE Prefs`" will force DAE to never use old cached information.

Starting with v6.0, DAE will query the Pro Tools folder for a file named "DaeOpts.txt". This file contains keyword switches to perform this functionality. Please refer to the section **Building & Testing Plug-Ins Without Relaunching Pro Tools** for more information.

Frequently Asked Questions

Q: I can see my plug-in's filename show up on the Pro Tools flash screen, but it's not showing up in the Pro Tools insert menus (or the AudioSuite menu.)

A: Ensure that your exports for the DLL, or the Shared Library, have been correctly defined. On the PC, `NewPlugIn`, and `_PI_GetRoutineDescriptor` should be exported, which is specified in the `.def` file. Under CodeWarrior, the Initilization, Main, and Termination Entry Points are defined in the project settings, under *PPC Linker*.

Q: I keep getting a kSubWidgetIndexRangeErr (-7125) when trying to instantiate my plug-in!

A: Have you recently altered the number or types of Type objects within your plug-in? If so, then DAE is still using the cached version of your plug-in and hasn't been updated with the information about your altered Types. Delete the contents of the "DAE Prefs" folder to force Pro Tools to refresh its plug-in cache.

Q: I'm break-pointing the method CProcess::*X* and it never gets called! What's going on?

A: Remember how calls are dispatched into the plug-in. Start at the top, breakpoint the appropriate call in `Dispatcher.cpp`, and trace the execution from there. In this manner, you can be sure that the call isn't terminating somewhere down the Group-Type-Process chain.

Q: I've added a DITL item to the UI, and instead of seeing the actual view I just see the static text of the item, e.g. "!SliderPict[('NoID') ('sld1') ()]"

A: The DITL item is not being properly parsed and recognized by the `CDialogView` class. Ensure that the view class is being properly registered at the Group level. In the SDK example plug-ins, this is always done within the Group level `Initialize()` method.

Q: I've added a background image to my plug-in. Everything's fine except when I click on the background all the foreground controls briefly flash away.

A: Be sure to disable the DITL item of the `CBackgroundPictView`.

Q: My control isn't responding properly to mouse input. It doesn't update until the mouse is released. What's going on?

A: Two possible things are occurring. One, the index of this control has inappropriately been defined as the *Master Bypass* control which Pro Tools/DAE treats differently. Ensure that the *Master Bypass* index is correctly set (to zero, if none exists). Two, there has not been a Control with a matching control ID added to the Control Manager - check your set of `AddControl()` calls in your Process.

Q: I'm implementing DoSetCursor() and I can see my cursor being updated for a brief period. However, it always flashes back to the standard arrow cursor.

A: Are you sure you are correctly returning `kFicPlugInDidSetCursor` for the `DoSetCursor()` method?

Q: I've added a control to my plug-in and defined it as being *automatable*. However, when I go to the automation list, the control does not appear.

A: You have probably defined page tables for this plug-in. In this case, the automation list is a reflection of the one-control-per-page generic ('`PgTL`') page table. Therefore, you also need to add the control to a page table. If a page table is not defined, the Plug-In Library will simply build a list of all your automatable controls for the Automation dialog.

Q: How can I dynamically change the range of a control? For example, how can I resize a set of radio group buttons based on the state of another control?

A: There really isn't a straightforward way to change the range of a control. The crux of the problem originates from automation and the interpretation of control values. Underneath it all, Pro Tools and DAE is dealing with signed 32-bit integers. A discrete control evenly maps its discrete set across the 32-bit range; therefore, if the number of items changed, so would the representation of those items.

Cross-Platform Considerations & Issues

Before becoming discouraged by the housekeeping involved in dealing with cross-platform development, keep in mind that once you get used to it, the process becomes quite simple. Also, if you

are going through the process of porting your plug-in from Mac to PC, Altura's tools will perform many of these steps automatically.

Line Termination Mac files use a simple carriage return for line termination; DOS files however, have a carriage return followed by a line feed. What does this mean to your plug-in? CodeWarrior IDE on Macintosh can handle both Macintosh and DOS line termination formats, while Microsoft Visual Studio on Windows cannot. Because of this, all cross-platform code files should contain DOS style line termination. In order to ensure that code files are saved in the DOS format, apply the global IDE setting in CodeWarrior to save all files with DOS line termination. This ensures that both the CW and Microsoft IDE will be capable of reading source code files.

A second problem with line termination occurs with the .asm files. Motorola's DSP assembler tools expect proper line termination on both platforms. Therefore, a file with DOS line termination will not be readable by the Macintosh version of the DSP tools. This leaves two options. (1) keep two versions of your DSP code handy - one with Mac termination and one with PC, or (2) do the conversion manually when taking the DSP code across platforms.

Resource Fork Macintosh resource files have the distinct “advantage” of being able to have both a resource and a data fork available to them. This architecture does not exist on the PC. As a result, when bringing a Mac file over to the PC any resource information will be lost. In order to compensate for this we need to preserve the resource info within the file's data fork. Also we want to be able to restore the resource fork when bringing a file back from the PC to the Mac. We have two tools available in the SDK, which allow for this process to occur: (1) `swap_resource_to_data.sh`, which will take a resource file's resource information and move it into the data fork, and (2) `swap_data_to_resource.sh`, which will reclaim the files resource information, and restore it into the resource fork. Both are destructive processes, overwriting any existing data in the “to” fork. Both are also shell scripts that can be run from the Terminal application in Mac OS X and take a file name as an argument.

***Update**: Mach-O .dpm bundle packages store their resource data in the Data Fork of their associated .rsrc file.*

You should have a version of the plug-in library on both Platforms, which will save you from having to do any resource swapping outside of your own plug-in code. If your plug-in is not running properly on both platforms, it is a good idea to verify that your resource file is in the correct format using ResEdit, or equivalent app.

ASCII characters Any ASCII character outside of the standard set of 128 may appear different on the Windows and Mac platforms. If you are using ASCII characters on one system that do not map to an appropriate character on the other, you can look for the closest match, or settle on another character. In many cases you may find it necessary to have a preprocessor conditional statement.

Note: The following characters are treated as special characters in Avid menus: ‘/’, ‘;’, ‘^’, ‘!’, ‘<’, ‘/’, ‘(‘. This is because Avid uses an older Mac/Altura function, `AppendStrings()`, to add strings to the menu. These characters should be avoided if you intend your plug-in for usage within Avid products.

Data Endianess Data on the x86 processors (PCs, Intel Based Macs) is stored in little-endian format (most significant byte is at higher memory address) unlike the PowerPC processor (ppc Macs), which stores data in big-endian format (Most significant byte of a word is at the lower memory address). In order to compensate for this, custom data stored within a plug-in resource must be byte-swapped when brought over to x86 processors. For standard resource data, Altura already provides the necessary mechanism for doing the byte swapping. The following is a list of resources that are automatically handled by Altura:

```
'acur', 'CDEF', 'CMAP', 'CURS', 'DRVR', 'FREF', 'ICN#', 'LDEF', 'PACK', 'PICT'
'SERD', 'STR#', 'WCTB', 'ALRT', 'cicn', 'CNTL', 'dctb', 'DSAT', 'FRSV', 'ICON'
'MBAR', 'PAT',  'pltt', 'sfnt', 'Styl', 'WDEF', 'ASID', 'clut', 'CODE', 'DITL'
'FKEY', 'FWID', 'INIT', 'MDEF', 'PAT#', 'ppat', 'SICN', 'TEXT', 'WIND', 'BNDL'
'CMAP', 'crsr', 'DLOG', 'FONT', 'icl8', 'INTL', 'MENU', 'PDEF', 'PREC', 'STR'
'vers'
```

For non-HCP protected versions of your plug-in, you need not worry about byte-swapping your DSP code stored in the SACD resource. This is because you will not use the SACD resource on Windows, but will rather assemble the code natively ensuring proper byte order. However, if your plug-in is an HCP plug-in and therefore is getting the SACD resource from the .rsr file, the Plugin Library will swap the SACD resources for you. If your plug-in has resources other than the ones mentioned above, you will be required to write a custom byte swapper. This is illustrated later in this chapter.

Pascal Strings Pascal string constants such as "`\pAny String`" are not compatible with the Microsoft compiler. Instead, you should hard code the length byte using an octal escape sequence. As an example "`\pThe String`" would get replaced with "`\012The String`".

Byte Swappers for Custom Resources If you use any special resources in your plug-in, you may need to add some code to your Windows version so that the bytes are handled properly when read from the `.rsr` file. The `.rsr` file contains the resources from the Mac and the bytes are preserved in the same "endian-ness" as on the PPC Mac (Intel based Macs are little endian). That is, on the Mac (ppc or x86), and in the `.rsr` file on XP, the resources are stored as big-endian.

The following considerations are only valid for development on the PC. Both ppc Macs and Intel based Macs handle resources in the same manner, without the need for special byte swapping routines.

At run-time on a PC, when resources are read from the `.rsr` file, each of the atomic elements of the resources need to be "swapped" to little-endian. This is what the byte swappers do -- they take a resource of a specific type and do whatever is necessary to get the resource to be "correct" in little-endian world. If your plug-in uses only "standard" resource types, you won't need to do anything -- the Altura code has default byte-swappers for all the known "standard" resource types. Otherwise, you'll need to "clone" an existing byte-swapper and register it during plug-in initialization. This is done in your `<plug-in name>Swap.cpp` file. For plug-ins that do not use any custom resources they can get away with using the following empty swap file provided in the SDK, `NullSwap.cpp`.

```
extern "C" void RegisterByteSwapper(void)
{
}

extern "C" void RemoveByteSwapper(void)
{
}
```

The following sample code demonstrates how to actually implement a byte-swapping routine on the fictitious Resource type 'dumb', where the 'dumb' resource only contains a single `long` value which needs to swapped.

```
#include "Mac2Win.H"
#include <Components.h>
#include "ASIExtrn.H"

extern char gResFile[];

void Swapdumb(char *swap)
{
     ASI_ByteSwapLONG((long *)swap);
}

#ifdef __cplusplus
extern "C"
#endif
F_PASCAL(BOOLEAN)
DefaultByteSwap( BOOLEAN Mac2Win, /*UWORD AppResIndex,*/ ResType theRT,
                   LONG lres_size, LPSTR lpdata );

F_PASCAL(BOOLEAN)
DefaultByteSwap( BOOLEAN Mac2Win, /*UWORD AppResIndex,*/ ResType theRT,
                   LONG lres_size, LPSTR lpdata )
{
     switch(theRT)
     {
          case 'dumb':
               Swapdump(lpdata);
               break;
          default:
               return false;
     }
     return true;
}
```

```
extern "C" ResType gl_AppResType[] =
{
     'dumb',
     0L
};

extern "C" void RegisterByteSwapper(void)
{
     ASI_RegisterCustomByteSwapper( gResFile, (ProcPtr) &DefaultByteSwap );
}

extern "C" void RemoveByteSwapper(void)
{
     ASI_UnregisterCustomByteSwapper( gResFile );
}
```

The global array `ResType gl_AppResType[]` should be initialized with your plug-in's custom resources, and null terminated. Instead of performing no operation, the `RegisterByteSwapper` and `RemoveByteSwapper` calls now call into Altura to register and unregister the plug-in's custom byte swapping routine, `DefaultByteSwap`. This byte swapping routine is associated with the plug-in's current resource file, which is acquired through the `gResFile` global variable. The switch statement of the `DefaultByteSwap` method could easily be extended to include other custom resource types.

In addition Altura provides the following set of helper functions for you to use with your byte swapping:

```
VOID ASI_ByteSwapWORD( UWORD *ptheWd )
VOID ASI_ByteSwapLONG( LONG *ptheLg )
VOID ASI_ByteSwapRect( Rect *ptheRect )
VOID ASI_ByteSwapLONGS( LONG *pData, LONG lSize )
VOID ASI_ByteSwapWORDS(UWORD *pData, LONG lSize )
VOID ASI_ByteSwapFLOAT( float *pFloat )
VOID ASI_ByteSwapFLOATS( float *pFloat , short size )
VOID ASI_ByteSwapDOUBLE( double *pDouble )
VOID ASI_ByteSwapEXTENDED( long double* pDouble )
VOID ASI_ByteSwap881( double881 *pDouble )
```

For additional help refer to Altura's `ASIBSwap.c` file.

Porting a Plug-In from Mac to PC

The following steps were derived from the process of porting SimplePlugIn, which is included in the SDK SamplePlugIns folder. (Although steps for DSP code cannot be seen in SimplePlugIn, as it has none.)

Step 1: Create a new project in Visual Studio
Select “Empty Project” as the type. This will automatically create a new solution for your plug-in

Step 2: Add your plug-in source files to the project

Step 3: Add the following files to the project:

- A byte-swapper: either DefaultSwap.cpp, DefaultHCPSwap.cpp, or your own custom byte-swapper
- DLLMain.cpp: Plug-ins are Regular DLLs and require thus require a DLLMain file
- YourPlugIn.def: This is a VC++ module-definition file, written by you. Look at the sample plug-ins for

MSDN reference: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_Module.2d.Definition_Files.asp

Step 4: Add the following libraries to your project

- DAE.lib
- DigiExt.lib
- DSI.lib

Your plug-in also may require one of the following libraries:

- DSPManager.lib

Take care to include both Debug and Release versions. Modify the property values to make sure that appropriate versions are not "Excluded from Build" and inappropriate versions are for each configuration.

Step 5: Add the plug-in library project to your solution

AlturaPorts\TDMPlugIns\PlugInLibrary\WinBuild\PlugInLib.vcproj

Step 6a: Add a new resource to your project

Select type "Version." Double-click on YourPlugIn.rc, and select VS_VERSION_INFO to edit. Make changes where necessary.

Step 6b (Optional): Add DSP Code

Add your .asm file to your project Resources folder. Right-click on YourPlugIn.rc and select "Properties." Add a custom build step to build the assembly code. Refer to SampleClick for an example of how this is done.

Step 6c (Optional): Add Page Table information

Refer to PeTE documentation for how-to.

Step 7: Convert your Mac resource file for PC

Make sure that the resource only uses the data fork, and not the resource fork. Fork swapping scripts exist within the SDK: DSPTools\bin\osx\swap_resource_to_data.sh & swap_data_to_resource.sh
There also exists the Xcode tool: AlturaPorts\TDMPlugIns\common\Mac\SwapForksTool\SwapForksTool.xcodeproj

Step 8: Modify the properties for your plug-in project

For the most part, you can mirror your properties to those of a sample plug-in, such as SimplePlugIn. However, make certain to pay special attention to the following settings (look to SimplePlugIn as a reference here):

- C/C++ -> Code Generation -> Struct Member Alignment
- C/C++ -> Advanced -> Forced Includes
- C/C++ -> Advanced -> Additional Options
- Linker -> General -> Output File
- Linker -> Input -> Module Definition File
- Custom Build Step -> General

While these aren't the only settings required to get your plug-in to build and run, they are some of the most crucial.

If you are having difficulties building your plug-in after this point, compare your plug-in to a working sample plug-in. The easiest way to notice any differences is to open the .vcproj file in a text editor outside of Visual Studio, or to look at the "Command Line" sections of the project properties while in Visual Studio.

Mac OS X Considerations & Issues

Transition to Mach-O / CFM Plug-In Compatibility

Pro Tools 7 will be the first Mach-O Pro Tools release. For the uninitiated, Mach-O is the preferred binary executable format for Mac OS X. This is a major change that all developers of plug-ins for Pro Tools should be aware of. Digidesign will be providing a temporary adapter (also known as a "shim") that will allow existing CFM plug-ins to run (with one notable exception -- see below), however we strongly recommend that all developers port their plug-ins to Mach-O now.

We have made every attempt to ensure that non-native CFM plug-ins function normally (via the shim), but we want to emphasize now that any future technical support related to CFM plug-ins and this shim will be de-emphasized. Additionally, future versions of Pro Tools released after Pro Tools 7 will not support CFM plug-ins, and will no longer include a CFM-to-Mach-O shim.

Please begin porting your plug-ins now! Porting is not difficult -- in most cases no source level changes are required. Once the process becomes familiar, the time required to port a plug-in may be measured in minutes rather than hours. Refer to the section "Porting a Plug-In to Mach-O" for a walk-through of the porting process in order to avoid common pitfalls and reduce porting time.

The Pro Tools 7 SDK will allow targets to be compiled for both CFM and Mach-O, as Mach-O plug-ins will be incompatible with pre-Pro Tools 7 versions of Pro Tools. Developers wishing to support pre-Pro Tools 7 versions of Pro Tools while moving their development environment to the Pro Tools 7 SDK will have to target both CFM and Mach-O, and ship both binaries. We expect to remove the CFM target in a subsequent version of the SDK as well, probably in the same timeframe as the shim.

Developers of MultiShell plug-ins, take special note: **CFM MultiShell plug-ins will not be supported by the shim, and must be ported to Mach-O**. This was due to some complexity in the interface between MuSh plug-ins and the DSP Manager that could not be practically addressed via the shim. Contact Developer Services if you have any additional questions.

Note that there is no break in compatibility and no porting required on Windows.

Porting a Plug-In to Mach-O

Introduction

To port plug-ins to Mach-O, it is required to use the Pro Tools 7 version of the SDK, along with Metrowerks CodeWarrior 9.5* **or** Xcode 2.2*. This guide is aimed towards those developing with CodeWarrior, and refers directly to the CodeWarrior IDE. For those wondering why this guide uses CodeWarrior instead of Apple's Xcode, the simple answer is that most plug-in development has used CodeWarrior, and an incremental shift to Mach-O would offer the least amount of code churn and risk for developers.

That being said, since the initial authoring of this guide, several important events affecting development on CodeWarrior have occurred. Most notably, Metrowerks sold their Intel compiler, and Apple announced they will be adopting Intel processors for future machines. Platform incompatibility is highly probable, and it is likely that future releases of the SDK (after Pro Tools 7) will not support CodeWarrior for this reason. Thus it will eventually be necessary to port your plug-in(s) to Xcode, which leaves two paths to consider when porting to Mach-O:

1) Follow this guide to port your plug-in(s) to Mach-O on CodeWarrior. This offers the smallest change to development environment, and will keep CFM -> Mach-O problems separate from any CodeWarrior -> GCC problems that may arise when you eventually port to Xcode. This also has the added benefit of having your CFM and Mach-O targets contained within the same CodeWarrior project. **NOTE: CFM plug-ins can only be built using CodeWarrior.**
2) Port directly to Xcode. This saves time, as porting to Xcode simultaneously ports your plug-in to Mach-O, and as stated before, you will eventually have to port your plug-in to Xcode. For more details on porting your plug-in to Xcode, refer to the accompanying "Xcode Porting Guide", available at http://developer.digidesign.com.

**As of the time of this publication.*

Getting Started

To convert an existing CFM based plug-in to Mach-O, open the CodeWarrior project for that plug-in, and create a new target for Mach-O. The fastest way to do this is to open a plug-in CW project, Control-

Click an existing CFM target, select “Create Target”, and then select the option “Clone existing target:” from the pop-up window to copy the CFM target settings for your new Mach-O target. This will populate your Mach-O target with many of the required source files, and reduces the amount of setup required for a Mach-O project.

The following sections explain how to modify the newly created target for Mach-O. Following these steps in order will ensure that all dependencies within the process are satisfied.

CodeWarrior Project Settings

After creating the Mach-O target, bring up its “Settings” dialog window. In this window, the following changes need to be made to the sections listed below. If an already ported project exists, refer to its settings, as it can serve as a template and expedite current porting efforts.

Note: The Template sample plug-in is a good reference for first-time porting. Template is already ported to Mach-O, as are all other Sample Plug-Ins within the Pro Tools 7 SDK.

- Target->Target Settings
  - Change **Linker** to “Mac OS X PowerPC Mach-O”

- Target->Access Paths
  - Check the box labeled: “Interpret foreign path separators”
  - Remove the following paths from **System Paths**
    - {Compiler}MSL
    - {Compiler}MacOS Support
  - Add the following paths to **System Paths**:
    - {Compiler}MacOS X Support
    - {Compiler}MSL/MSL_C
    - {Compiler}MSL/MSL_C++
    - {OS X Volume}Developer/Headers/FlatCarbon
    - {OS X Volume}usr/include
    - {OS X Volume}usr/lib

    Make certain the paths are in this order! Otherwise, the compiler will generate build errors.

*Note: If an already ported project exists, it may be easier to simply drag and drop these paths into the **System Paths** box. However, take care the path order and relative paths (e.g. {OS X Volume}) remain consistent.*

- Target->PPC Mac OS X Target
  - Change the **Project Type** to “Bundle Package”
  - Choose your bundle name, and change the extension from .bundle to .dpm
  - Change **Creator** from “????” to ”PTuL”
  - Change **Type** to “TDMw”

- Language Settings->C/C++ Settings
  - Make certain that the box labeled “ANSI Strict” is not checked

- Language Settings->C/C++ Preprocessor
  - Make “#include <MSLCarbonPrefix.h>” the first line of the **Prefix Text**
  - Check the box labeled: “Use prefix text in precompiled headers”
  - Precompiled Headers: If the project will contain targets for both CFM and Mach-O, it is necessary to compile a corresponding version of all precompiled headers separately for use with each ABI. A good way to handle the generation of these separate versions is in the precompiled header with preprocessor directives and pragmas:

    #if __MACH__

```
        #pragma precompile_target "SampleHeaderPPC_MachO++"
#else
        #pragma precompile_target "SampleHeaderPPC++"
```

This will create a different precompiled header for each binary format, and ensure that your project doesn't build with an incorrectly formatted precompiled header file.

*Note: If renaming any precompiled header, don't forget to alter the **Prefix Text** to reflect any name changes.*

- Linker->PPC Mac OS X Linker
  - Change **Export Symbols** to "Use #pragma" or "Use #pragma and '.exp' file" if your plug-in uses an .exp file
  - Make sure that the **Main Entry Point** field is empty
  - [Optional] For Debug builds, check the box labeled: "Generate SYM File"

Frameworks

After altering the settings, save and return to the project window. The next step is to add the necessary frameworks to the Mach-O target. The two required frameworks are: Carbon.framework and System.framework. These can be easily added by opening a Finder window to /System/Library/Frameworks and dragging the necessary frameworks onto the plug-in project.

Libraries

Next, static libraries compiled for CFM will need to be unlinked from the Mach-O target, and replaced with libraries compiled for Mach-O. Moving forward, Digidesign will provide versions of PluginLib for both Mach-O and CFM legacy support. The two versions will be demarcated as follows: Mach-O: PluginLib-Dbg.lib / CFM: Plugin_CFM-Dbg.lib

The following libraries, if present, need to be unlinked from the Mach-O target:

MSL_C++_PPC.Lib
MSL_C_Carbon.Lib
MSL_Runtime_PPC.Lib
CarbonLib
PluginLib_CFM-Dbg.lib
PluginLib_CFM-Rel.lib
RTASClientLib-Dbg.lib
RTASClientLib-Rel.lib

The following libraries need to be added to the project, and/or linked to the Mach-O target. Remember, this can most easily be accomplished by dragging and dropping from an existing Mach-O project.

/usr/lib/bundle1.o
PluginLib-Dbg.lib
PluginLib-Rel.lib

Note: The reason behind the conspicuous absence of a Mach-O version of RTASClientLib is that RTASClientLib sources are now compiled directly into the PluginLib library.

** These are Digidesign modified builds of the standard CodeWarrior MSL libraries.*

Property List Files

Next, because the plug-in is now a bundle package, it is necessary to generate a plist file to include within the package. CodeWarrior will generate the Info.plist file, but requires two files to do so: a plist compiler file and a corresponding header file. Please refer to the Template sample plug-in for examples of both. In the Template/MacBuild directory, these files are named Template_info.plc and TemplateBundle.h, respectively. Customize these files for the plug-in by replacing all instances of "Template" with the plug-in name, and renaming the files accordingly. (e.g. If porting a plug-in named YourPlugIn, you would name them: YourPlugIn_info.plc and YourPlugInBundle.h and replace all occurrences of "Template" in the two files with "YourPlugIn") But pay special attention to the *BUNDLE_ID in the *Bundle.h file. This ID is specific to the plug-in binary, and follows Apple's recommended "Java-style" naming scheme. Take care in selecting a name, as it must be unique amongst all potential Digidesign and 3rd party plug-ins. The suggested naming scheme is to combine manufacturer name and plug-in name to create BUNDLE_ID. (e.g. com.yourcompanyname.YourPlugInName) Once all modifications are complete, include YourPlugIn_info.plc in the YourPlugIn project, and make sure it is a target for Mach-O.

Targets

The final step is to ensure that the Mach-O target is targeting the appropriate PluginLib libraries. Go to the "Targets" tab in the Project Window, and make any necessary changes so that the Mach-O target includes the Mach-O libraries and not the CFM libraries as targets.

At this point, the plug-in should be ready to build for Mach-O. If you encounter any build errors, please refer to the "Common Errors" section below.

Plug-In Bundle Format

Plug-in bundles should use a .dpm filename extension.

Plug-in bundles should have creator: 'PTul' and type: 'TDMw'.
These properties are set in Codewarrior's 'Target->PPC Mac OS X Target' preference pane, and you may check them by verifying that the PkgInfo file in your plug-in bundle contains the string: TDMwPTul Pro Tools currently does not load plug-ins that do not follow this format. Note that these settings will cause your plug-in to use the standard Pro Tools plug-in icon unless you implement a custom icon.

The MicrobeSampler sample plug-in demonstrates how to create a custom icon for a Mach-O plug-in.
Please view README.txt in the MacBuild/MicrobeSamplerCustomIcon/ directory of this plug-in for more information.

Pro Tools supports using a single bundle for both Mach-O and CFM versions of a plug-in.
To make plug-in deployment easier on the developer, Pro Tools allows for both the legacy CFM version of the plug-in and the Mach-O version to be combined into the same plug-in bundle. *Both the Mach-O plug-in and the CFM plug-in should be compiled and Pace wrapped completely and independently as if making two separate plug-ins*. Then the complete CFM plug-in may be placed in a 'CFMPro ToolsPlugIn' folder inside the Mach-O plug-in's bundle, in the Contents folder. CFM plug-ins in the 'YourPlugIn.dpm/Contents/CFMPro ToolsPlugIn/' folder will be loaded on versions of Pro Tools prior to 7.0, while Pro Tools 7.0 will load the Mach-O version of your plug-in. The 'Template Debug Bundle Merge' target in the Template sample plug-in provides an example of how to automate this process into the Codewarrior build environment.

Common Errors

Error: Multiple link errors pertaining to undefined code, particularly pthreads. Typically, the "Too many link errors" message is present as well.

Solution: Check that the Carbon.framework and System.framework are included and linked to all Mach-O targets

Error: Link error: undefined: 'start' (code)
Solution: Ensure that Main Entry Point field is empty under Settings->Linker->PPC Mac OS X Linker

Error: "preprocessor #error directive" ... "You must have the non-MSL C header file access path before the MSL access path"
Solution: Check that "Include Prefix Text" is checked under Settings-> Language Settings->C/C++ Preprocessor and that all System Paths are in the correct order.

Error: Multiple "illegal expression" and/or "illegal declaration" errors where the code appears valid
Solution: Double check that "ANSI Strict" is unchecked under Settings->Language Settings->C/C++ Language

Error: Link Error: Output file would contain no segments, check export settings
Solution: Double check that Export Symbols has been set to either "Use #pragma" or "Use #pragma and '.exp' file" under Settings->Linker->PPC Mac OS X Linker

Error: Runtime error: Pro Tools is crashing when loading a plug-in.
Solution: This is often the result of an illegal or absent Info.plst file. Make certain the plug-in has both a .plc and header file, that the .plc file is included in the plug-in project, and a target for Mach-O.

Special Cautions

If your plug-in is built using the Metroworks PowerPlant framework, note that PowerPlant is incompatible with Mach-O. It is possible to use the PowerPlant X framework with Mach-O, but several functions from PowerPlant have been deprecated, so special attention to porting may be required.

Any code that directly manipulates the plug-in's FSSpec must be amended to refer to the plug-in's bundle identifier instead. This can be accomplished by using the `UseResourceFile()` and `RestoreResourceFile()` utility functions.

Apple Mach-O References

The Apple Carbon Mach-O Porting Guide:
http://developer.apple.com/documentation/Carbon/Conceptual/carbon_porting_guide/cpg_buildstruct/chapter_3_section_5.htm

Introduction to the Mach-O Runtime Architecture:
http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/index.html

Description of Bundle Packages:
http://developer.apple.com/documentation/CoreFoundation/Reference/CFBundleRef/CFBundleRef.pdf

CodeWarrior 9.5 and Mach-O Debugging Issues

The CodeWarrior 9.x debugger can be erratic when working with Mach-O projects. This most often results in CW missing/timing out on breakpoints. These are known issues, and you should take the following into account when debugging Mach-O projects using CodeWarrior:

1) Sometimes when adding or removing a breakpoint in a Mach-O component, Codewarrior will cache all of the code files in the project, which can take up to 45 seconds or so.

2) When stepping through Mach-O code, you must step quickly or Codewarrior will timeout and you have to press "play" in the debugger to keep going, so try to use breakpoints exclusively.

Porting Plug-in Projects to Xcode

This porting guide is designed to assist plug-in developers port their plug-in projects from Codewarrior 9.x to Xcode. However, this guide is not meant to be a comprehensive guide to Xcode. Basic Xcode knowledge is assumed. Refer to the documentation at http://developer.apple.com/tools/xcode/ to familiarize yourself with Xcode's features and usage. This guide is written for and requires all three of the following: Mac OS 10.4.x, Xcode 2.1, and Plug-In SDK 7.0.

The porting process described in this guide uses a skeleton project. Basing your plug-in project on this skeleton project will alleviate much of the set-up work that is required to build a plug-in. There is some tweaking that must be done, but most of this has to do with adding project-specific files and managing paths to dependencies. The skeleton project follows some conventions that are listed below. These conventions were originally designed to make managing and updating of our internal projects easier. These conventions are suggestions to you, the developer, but they are not hard and fast rules. Your mileage may vary.

Suggested Plug-In Project Conventions

- The skeleton project uses a common "configuration file", a text file which lists shared project settings.
- Debug Targets should use TDMPlugIns/common/Mac/CommonDebugSettings.xcconfig
- Release Targets should use TDMPlugIns/common/Mac/CommonReleaseSettings.xcconfig
- These config files takes care of a couple other conventions that plug-ins should follow:
    - debug targets should define _DEBUG to 1 in project settings
    - Plug-ins should output intermediates to MacBag/Intermediates
    - Plug-ins should output precompiled headers to MacBag/Intermediates/SharedPrecompiledHeaders
- A common pre-compiled header is supplied in the SDK. A plug-in can use this header or define a custom one. The supplied header is AlturaPorts/TDMPlugIns/Common/Mac/CommonPlugInHeaders.pch. The xcconfig files set this up to be the default header used.
- All paths specified should be relative to the SYMROOT env variable. This is so that plug-ins can more easily be moved to different directories -- paths aren't hard-coded.
- Assumes that MacBag will always be located in the root directory of the SDK.
- "Run Script" build phases should always use the SYMROOT env variable when specifying paths: e.g. mkdir -p $SYMROOT/Debug/Plug-Ins
- Plug-ins should have separate Debug and Release targets, rather than separate configurations. This is because for Debug and Release to include different files (for example different static libs, like the plug-in lib) you must use targets, you can't use configurations. In addition, there should be a single configuration called "Components". This is so that all products will get built to MacBag/Components.
- Plug-ins should build to the MacBag/Components directory, and then a "Run Script" build phase should be added that copies the plug-in to the proper directory -- this is because Xcode always creates an output directory named after the active configuration.

Step-by Step Porting Instructions

1. **Copy** AlturaPorts/TDMPlugins/SamplePlugins/XCodeProjectTemplateFolder/ProjectSkeleton.xcodeproj to the MacBuild folder of your plug-in project.
2. **Rename** the ProjectSkeleton.xcodeproj file to reflect current plug-in project.
3. **Open** the renamed project in Xcode.
4. **Set the active configuration to 'Components'.**
5. **Fix any broken paths** (any file in red text except "Products")
    a. In the Groups & Files pane, select both .**xcconfig** files
        i. File -> Get Info -> General Tab -> Path Choose
        ii. Set path to AlturaPorts/TDMPlugIns/common/Mac/
        iii. These files should now be displayed in black text.
        iv. You may want to open these config files to familiarize yourself with their contents.
    b. Select **PlugInLibrary.xcodeproj**
        i. File -> Get Info -> General Tab -> Path Choose
        ii. Set path to AlturaPorts/TDMPlugIns/PlugInLibrary/MacBuild/PlugInLibrary.xcodeproj
        iii. This file should now be displayed in black.
        iv. Make sure PluginLibrary products are added to appropriate targets
6. **Add appropriate PlugInLibrary dependency** to each Target
    a. Select the **Debug** target in the Groups & Files pane, then choose File -> Get Info
        i. In the General tab, click the '+' sign to add a dependency
        ii. Add "Plugin Library **Debug**"
    b. Select the **Release** target in Groups & Files pane, then choose File -> Get Info
        i. In the General tab, click + sign to add dependency
        ii. Add "Plugin Library **Release**"
7. For each Target, **update appropriate properties**
    a. Select **Debug** target in Groups & Files pane, choose File -> Get Info
    b. In the Build tab, notice that the build setting are based on the **CommonDebugSettings.xconfig** file
    c. Modify the "**Development Build Products Path**" to point to the MacBag folder in the SDK root directory. This path should be **relative to** the xcodeproj file. **IMPORTANT**: Make sure this is correct as this sets the **SYMROOT** environment variable that many other settings are based upon.
    d. Modify the "**Product Name**" to suit your plug-in. Do *not* include the .dpm extension.
    e. **Repeat** a-d for the **Release** target, except notice that *b* is based off of **CommonReleaseSettings.xconfig**

8. **The xcconfig files define a pre-compiled header that is used common to all plug-ins.** If you find that you need to define certain directives that would normally be put in a pre-compiled header, you can add these to your targets build properties using the "**Preprocessor Macros**" build setting for each target. Or if you need to use a custom pre-compiled header, you can specify the file to use in the "**Prefix Header**" build setting for each target. Be sure to include the common prefix header in your custom file for consistency. See Suggested Conventions above.

9. **Add existing source and header files**. Refer to your existing Codewarrior project for guidance on what files are needed. Be sure to add these files "**Relative to Project**" and be sure to **uncheck** "Copy items into...". Be sure to check all targets that apply. If any source code changes need to be made to accommodate different build environments, i.e. Codewarrior v. Xcode, one approach is to use #ifdef's with the appropriate compiler ID, __MWERKS__ and __GNUC__ respectively.

10. **Add resource files to project**. Again refer to your existing Codewarrior project for guidance. Be sure to add SACD.r files. You no longer need Codewarrior .plc files, as Xcode generates the plug-in's .plist file using a different method. Inspect the Get Info pane for .rsr and .rsrc files to make sure they are listed as File Type "archive.rsrc". **NOTE: Resource files with a .rsr extension will not be recognized as file type "archive.rsrc". They must be explicitly changed to have this file type.**

11. Before building, you must use the **SwapForksTool** (AlturaPorts/TDMPlugIns/common/Mac/SwapForksTool/) to **swap all resources to the data fork**. This is required due to a bug in Xcode's Rez utility that requires the input resources to exist in the data fork, when the output is written to a data fork file. SwapForksTool is simply an Xcode project that runs a script, which searches and swaps the forks for all .rsr files in the TDMPlugins folder. Simply change the target to SwapResourceToData or SwapDataToResource and click build.

12. **Add a .plist file to the project**, but don't include it in any targets. In each target's build settings (Get Info >Build tab) set the "Info.plist" setting to point to the appropriate plist file for that target. Different .plists may be required for each target. Although, Debug and Release targets could share a common Info.plist file. Executable name and Bundle name within this plist can change dynamically by specifying $(PRODUCT_NAME). For Pro Tools to properly recognize and load your plug-in, the plug-in's plist file must specify 'TDMw' as PackageType/Type and 'PTul' as BundleSignature/Creator. This will also allow the plug-in to display the default plug-in icon defined by Pro Tools. Alternatively, a plug-in can have a custom icon. See the Staley Technical Briefing Document for info on how to use a custom icon. See the Sample Plug-Ins or the Apple developer website for more info on plists.

13. **Inspect Target Build phases**: Expand the Debug/Release targets in the "Groups and Files" pane.

    a. **Build DSP Code**: If the plug-in has DSP code to be built, this build phase called will take care of this step. If building of DSP code is not required (non-TDM plug-ins), then delete this Build phase from both targets.

    b. **Compile Sources**:

    c. **Link Binary with Libraries**: Make sure that all required Frameworks and libraries are included in this step. Most plug-ins will not need to change this stage.

    d. **Build Resources**: Make sure all relevant .r, .rsr, and .rsrc files are listed in this build phase.

    e. **Copy to Plug-Ins Directory**: This takes care of copying the final .dpm plug-in from the Build Products location to the proper Plug-Ins folder in MacBag. This is set up for you and should not need any tweaking unless you require that the file be copied elsewhere. Or you can delete this step if you do not need the file copied as part of the build process

14. You should be ready to build. Give it a try.

Known Issues with Using Xcode

1. **Build products in MacBag/Development folder instead of MacBag/Components folder:** As recommended in step 4 above, you should explicitly set the active build configuration to "Components" before you build your project. As in the sample plug-ins, if the only available build configuration is user defined, i.e. Components, Xcode does not implicitly select this configuration upon initial start-up. It must be manually selected in the Build window. Once this configuration is selected the first time, it will persist in subsequent uses of the project. This is important for two reasons. First, the settings associated with the "Components" configuration will not be applied if it is not the active build configuration. Also, this will cause the build products to be built in the $(SYMROOT)/Development folder instead of the $(SYMROOT)/Components folder, possibly confusing search paths further down the process.
2. **Spurious occurrences of headers not being found**: If you inspect the header search paths supplied by the xcconfig files, you will see certain paths suffixed with two asterisks, i.e. AlturaPorts/Fic/Interfaces/**. This instructs Xcode to recursively search this folder and any of its subfolders for header files. As experienced while constructing this SDK, Xcode does not always expand these search paths to search recursively according to the expected behavior. As such, upon building standard projects, i.e. the PlugInLibrary, build errors may occur due to headers not being found. A simple workaround is to keep trying a multiple times and eventually Xcode will expand the search paths properly. Another solution is to close the project and re-open it.
3. **Resource Build stage fails with "fnferr"**: An "fnferr" can be returned from the "Build Resources" build stage of a project due to an incompatibility between Xcode and Rez. When Xcode invokes Rez to build the resources of a project, it passes in the path as a concatenation of the absolute path of the project folder, the path defined by the SYMROOT environment variable, and the hard-coded relative path to where it wants the output of Rez to go. This path can be quite long depending on where your project file is on your file system and how long the relative path is to SYMROOT. Unfortunately, Rez poorly handles path/filenames of length 256 characters or greater. In fact it truncates them to 255 characters. This can lead to a discrepancy of the actual path/filename generated by Rez and the path/filename that Xcode expects to find, thus causing the "file not found error". This is an issue for such sample plug-ins as MicrobeSampler and Template_NoUI, as they have somewhat longer filenames and reside in a deep directory structure within the SDK. This should not be much of a problem for most plug-ins if
    a. The plug-in projects are at the typical level directly within the TDMPlugins folder in the SDK (or higher)
    b. The absolute path to the SDK is not too long.
    c. The SYMROOT path is not obnoxiously long.

    If you experience problems with this error, try adjusting the folder depth of your project or adjusting the three points above. One quick method is to rename the somewhat lengthy name of the root SDK folder to something short, like "SDK".
4. **Project settings not being saved with project:** Xcode projects (.xcodeproj files) are actually bundles that contain numerous other files including a project.pbxproj file which is the actual xml-like file that defines your project settings. Due to varying factors, i.e. SCM, these files inside the bundle can become locked (read only), yet the enclosing bundle does not reflect the locked state. Xcode will still open these locked projects and you can make changes to the settings and build with these changes so long as you do not close the project. If you close a locked project, all settings will be lost. All files supplied in the SDK are un-locked. But if you ever change the locked state, be vigilant when updating or changing you project settings. Make sure you don't lose your changes.

DSP Code Development Tools

In order to automate the DSP code building process, several scripts were developed which make DSP code building very simple for the developer. Basically, the same scripts are used on both Mac and PC, with the pertinent modifications.

To use the DSP build system, you need two things:

1 A directory called `DSP` within your plug-in's root folder, containing the file `makefile_plug.gmk` and your DSP code `.asm` files. Your root folder should be located in `/AlturaPorts/TDMPlugIns/`.

2 The `DSPTools/bin` directory and its contents. To be even more specific, all you should need is `DSPTools/bin/scripts` and `DSPTools/bin/`*`platform`* where *`platform`* is either `win` or `mac`, whichever you want to build on.

Macintosh

To build DSP code on the Mac, the general idea is:

In your `makefile_plug.gmk` file inside your Plug-In's `DSP` directory, you specify the following variables:

This variable tells the system which is your plug-in's root folder.

```
Plugin := <YourPlugInFolder>
```

This variable sets up all the .asm files which your want to compile for inclusion in your plug-in.

```
PluginFiles := <ASMFile1> <ASMFile2> ...
```

This variable tells the system which DSPs you are targeting, that is, it will use different Motorola assembler programs for the different DSPs to generate the correct object code for each .asm file:

```
DSPs := M O P G2
```

Where `M` = Merle (56002), `O` = Onyx (56301), `P` = Presto, G2 = 56321

Now, you must update the *`SACD.r` file found in the MacBuild folder. This file maps the final output files to an associated "SACD" ID number. The scripts use this file to create the final `.rsr` file that is compiled into the plug-in.

Comment [MSOffice2]: I'm not sure about this step.

Now, simply include the DSP build script (DSPTools/bin/osx/makeDSP_OSX.pl) in your CodeWarrior project along with the *SACD.r file. On Xcode, include your *SACD.r file in the project, and if you based your plug-in on the ProjectSkeleton.xcodeproj, the Build DSP Code step under each target should already include the DSP build script.

Comment [MSOffice3]: This is out of date. To be removed before posting.

Windows

In general, DSP code is built automatically as a custom build rule in the Visual Studio Project.

In your `<PlugIn>.rc` file, you specify the Custom Build step as (select the file, right click and select Settings, Custom Build tab):

Commands:

```
echo off
call ..\..\common\make\make.bat
```

```
echo on
rc -r -fo $(IntDir)\$(InputName).res $(InputName).rc
```

Outputs:
```
$(IntDir)\$(InputName).res
dummy
```

In addition, the `.rc` file should contain "includes" pointing to the `DSPCode.rcx` file. Open the `.rc` file as a text file to verify this is true. This information does not naturally get displayed within Visual Studio.

The file `DSPCode.rcx` must be in your `<PlugInDir>/WinBuild/` folder, and it should specify the DSP code object as resources:

```
#define IDR_VOLUMEPROCESSONYX           3

IDR_VOLUMEPROCESSONYX             DSPCODE DISCARDABLE
BEGIN
#include "DSP\out\VolumeProcess.O.U.56k"
END
```

Notice the path of the `.56k` file is the actual binary code generated for your DSP code.

Inhibiting the Production of Intermediate Files

If you wish to inhibit the production of `.cld`, `.lod`, `.lst`, and `.map` files, set the environment variable `digi_output_lod_lst_map` to `false`, from the Command Prompt. Somehow "`cld`" was left out of this name, but they are affected too.

Directory structure must match the SDK

The DSP build scripts are written with the design objective of minimizing the need to configure your environment. You must have a directory structure identical to that presented in the SDK since the scripts rely on this structure to "get around" via relative paths.

Part VI: Sample Plug-Ins

Chapter 13: Sample Plug-Ins Overview

Digidesign provides a suite of sample plug-ins for our 3rd party community to use as examples during development. We try to make sure that each of our major technologies and features are demonstrated in at least one example. Below are some charts that will help you to understand which of our sample plug-ins demonstrate which features.

Technologies

Sample Plug-In	TDM	MuSh	AS	RTAS (AS-adaptor)	True RTAS	Onyx (MIX)	Presto (HD)	321 (HDIAccel)
Template	X	X	X		X	X	X	X
Template_DMA	X							X
Template_NoUI					X			
SimplePlugIn					X			
Microbe	X			X		X	X	X
MicrobeSampler					X			
56kTdm2	X						X	X
SampleClick		X		X		X	X	X
RDH			X					

Features

Sample Plug-In	DirectMidi	External Meter / Internal Clip	Aux Output Stems	Non-Digi User Interface	DMA (DSP-to-ASIC)	DMA (Host-to-DSP)	5.1-to-stereo / LCRS-to-stereo	Host Instrument Optimizations
Template		X						
Template_DMA		X				X		
Template_NoUI				X				
Microbe	X							
MicrobeSampler	X							X
56kTdm2					X		X	
SampleClick	X		X					
RDH								

Chapter 14: The Template Plug-In

The Template Plug-in is a simple plug-in which implements a gain "algorithm" in several different Types. The Template demonstrates a mono AudioSuite Type, a mono, stereo, and 5.1 surround RTAS Type, a mono non-MuSh TDM Type, and lastly, a mono MultiShell Type. The MultiShell and non-MultiShell plug-ins are mutually exclusive and built selectively depending on the Project's Target settings.

Though basic, the Template still exercises quite a bit of plug-in functionality, and is a very good starting point for creating a new plug-in.

Making the Template RTAS-Only

If you're using the Template as your starting point and only developing an RTAS plug-in, you will likely wish to remove the Non-MuSh and MuSh TDM Types. The following steps will do the trick:

1 Remove the files `CTemplateProcessMuSh.cpp/.h`, `CTemplateProcessTDM.cpp/.h`, and `TemplateSACD.r` from the project.

2 Modify `CTemplateGroup.h` such that the class `CTemplateGroup` inherits from `CEffectGroup`, rather than `CEffectGroupTDM` or `CEffectGroupMuSh`.

3 Modify `CTemplateGroup.cpp` by removing the `DefineDspResourceAndMaxChannels()` calls from the constructor of `CTemplateGroup`.

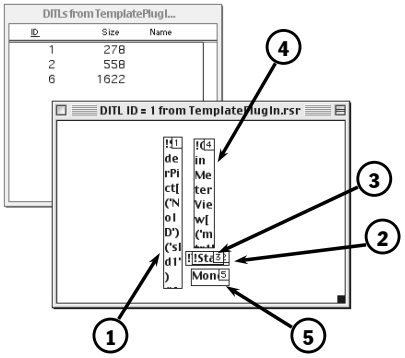
4 In `CTemplateGroup::CreateEffectTypes()`, remove the specification blocks for the MuSh and Non-MuSh TDM Types. In other words, remove all code between `#ifdef TEMPLATE_IS_MUSH` and the enclosing `#endif`.

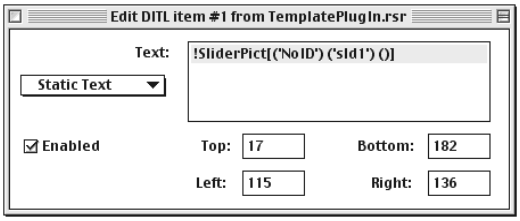
The GUI Resources

The Template has three DITL Resources, one each for the mono, stereo, and 5.1 surround Types. They have DITL IDs 1, 2, and 6, respectively.

The screenshot here shows the DITL layout of the mono Type. There are five "Static Text" elements that are used to define the dialog window. Let's examine them in order.

Note that item 2 is partially masked since it is behind item 3.

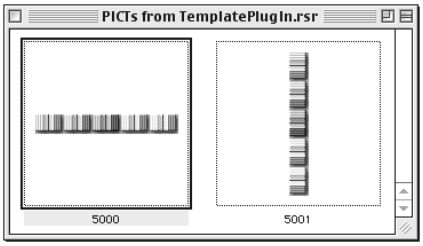




(1) Double-clicking item #1 opens the editor for the element. Here the rectangular dimensions and the text contents can be manually edited. Note, that the element should be checked *Enabled*.

At run-time, the text of the DITL item, `!SliderPict[('NoID') ('sld1') ()]`, is parsed by a `CDialogView` within the Process, providing a definition of the view class to be instantiated. The chapter **The Graphical User Interface** details this system in more detail.

Since the call `CSliderPictControlEditor::RegisterView()` was issued at the Group level, the view system knows to instantiate a `CSliderPictControlEditor` from the identifier "`!SliderPict`".

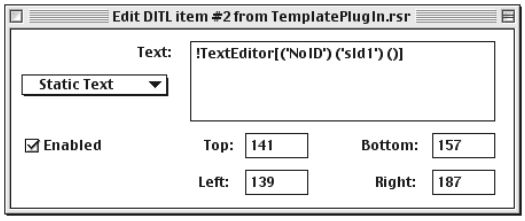


This view class implements a "gain fader" using a graphical picture for the knob. The `'NoID'` identifier is a dummy identifier for the element. The second ID, `'sld1'`, is critical; this ID links the view element to other view elements (like DITL item #2, as we will see next). It also provides the communication linkage to the Process.

The last empty set of brackets could possibly include keywords that further define the view.

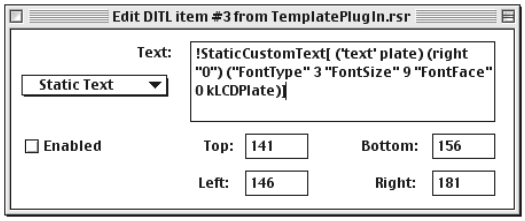
Lastly, for this view element to function properly, two PICT elements must be added to the Resource. These pictures contain the knobs, with color highlighting, that the slider uses. They are hard coded to be have IDs 5000 and 5001.

(2) Item #2 defines a `CTextControlEditor`, a view class that is registered by default. This element shares the same ID, `'sld1'`, with the `SliderPict` element. This links the text editor with the slider back to the Control Manager. This way, moving the slider will simultaneously update the gain value displayed in the text editor. Likewise, manually entering a value in the text editor will update the slider.

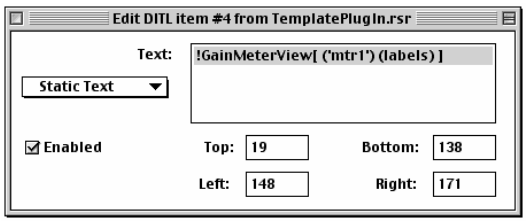


Notice that the rectangle of this item surrounds that of item #3. This is because `CTextControlEditor` only provides editing capabilities, and must wrap a text display class that has a view ID of `'text'`.

(3) The text display class is `CStaticCustomText` and is invoked by the "`!StaticCustromText`" identifier. Note, that the first parameter is the required `'text'` ID. The remaining parameters define the font and style of the text box. The *Enabled* box is intentionally left unchecked since it is controlled by the `CTextControlEditor`.



④ To display the volume level output by the Process, a meter view class is used. This is represented by DITL item #4 and has a view ID of `'mtr1'` The *labels* keyword causes the view to display dB values next to the meter "LEDs". The instantiated class is `CGainMeterView`.



⑤ The remaining DITL item is simply a static text item displaying "Mono".

Connecting the Process to the GUI

The interaction of the Process with the GUI window occurs in a handful of methods.

■ The first is `CTemplateProcess::EffectInit()` which is called once immediately after the Process has been instantiated.

```
void CTemplateProcess::EffectInit()
{
     AddControl(new CPluginControl_OnOff('bypa', "Master Bypass", false, true)); // Default to off
     DefineMasterBypassControlIndex(1);

     // DLOG Ids are conveniently equal to the number of channels they display:
     this->DefineDLOGId(GetNumOutputs());  // DLOG Ids = 1(Mono), 2(Stereo), and 6(5.1)

     switch (GetOutputStemFormat()) {
          case(ePlugIn_StemFormat_Mono):
               AddControl(new CPluginControl_LinearGain('sld1', "Mono Gain\nGain\nGan", 0.0, 1.0, 0.1, 1.0, true));
               break;
          case(ePlugIn_StemFormat_Stereo):
               AddControl(new CPluginControl_LinearGain('sld1', "Left Gain\nLeft\nL", 0.0, 1.0, 0.01, 1.0, true));
               AddControl(new CPluginControl_LinearGain('sld2', "Right Gain\nRite\nR", 0.0, 1.0, 0.01, 1.0, true));
               break;
          case(ePlugIn_StemFormat_5dot1):
               AddControl(new CPluginControl_LinearGain
                                   ('sld1', "Left Gain\nLeft\nL\n", 0.0, 1.0, 0.01, 1.0, true));
               AddControl(new CPluginControl_LinearGain
                                   ('sld2', "Center Gain\nCenter\nCen",0.0, 1.0, 0.01, 1.0, true));
               AddControl(new CPluginControl_LinearGain
                                   ('sld3', "Right Gain\nRite\nR", 0.0, 1.0, 0.01, 1.0, true));
               AddControl(new CPluginControl_LinearGain
                                   ('sld4', "Left Surround Gain\nLeftSR\nLSR", 0.0, 1.0, 0.01, 1.0, true));
               AddControl(new CPluginControl_LinearGain
                                   ('sld5', "Right Surround Gain\nRiteSR\nRSR", 0.0, 1.0, 0.01, 1.0, true));
               AddControl(new CPluginControl_LinearGain
                                   ('sld6', "LFE Gain\nLFE", 0.0, 1.0, 0.01, 1.0, true));
               break;
          }
}
```

In `EffectInit()` two main things are accomplished. The appropriate DLOG ID is selected based on the number of outputs the Process has. Also, the correct controls are added to the Control Manager, again, based on the number of outputs. (Another less dynamic approach might involve implementing multiple Process classes corresponding to each output Type. Then within each Process class, individually, the appropriate controls could be implemented and the correct DLOG ID specified.)

■ The next place interaction with the GUI occurs is in the method `SetViewPort()`. This method is called every time the Process is attached to (or detached from) a window.

```
void CTemplateProcess::SetViewPort (GrafPtr aPort)
{
     using namespace TEMPLATE;
     // Always start with inherited, in order to attach control to view and init view.
     CEffectProcess::SetViewPort(aPort);

     if (GetPlugInView() != NULL) {
          for (int i = 0; i < GetNumOutputs(); i++) {
               if (i != kMeter51_LFE) {
                    mGainMeterView[i]
                         = dynamic_cast<CGainMeterView *>(GetPlugInView()->FindSubView(TEMPLATE::kMeterIDs[i]));
               }
          }
     }
}
```

Here, pointers to the meter view classes are found. At this point in time, the meter view classes have been instantiated behind the scenes by the Plug-In Library from parsing the DLOG/DITL resources. They are placed in the member array `mGainMeterView` for quick access during runtime.

■ During processing time, the meters must be explicitly updated with new output values. This updating originates in the method `DoTokenIdle()`. This method is invoked on a periodic basis by DAE to handle the token system (SDS) so that controls get visually updated. By overriding this method -- but still calling the inherited method -- we can utilize this periodic pulse so that the meters can also be updated. Note, this call is only invoked when the plug-in's window is open.

```
void CTemplateProcess::DoTokenIdle (void)
{
     CEffectProcess::DoTokenIdle();     // call inherited to get tokens so we can move controls.

     // Disable metering for AS during Preview mode because doidle is called too frequently during
     // preview when the mouse is moving.
     if(!IsAS())
          UpdateMeters();
     else if (GetASPreviewState() == previewState_Off)
          UpdateMeters();
}
```

This method more or less directly calls the Template method `UpdateMeters()`, with some additional intelligence to circumvent limitations of AudioSuite's preview mode.

```
void CTemplateProcess::UpdateMeters(void)
{
     using namespace TEMPLATE;
     // update meter views with the current values
     long Ticks = TickCount();
     if (Ticks != mLastMeterTicks)
     {
          mLastMeterTicks = Ticks;

          for (int i = 0; i < GetNumOutputs(); i++) {
               if (i != kMeter51_LFE) {
                    if (mGainMeterView[i])
                         mGainMeterView[i]->SetValue( GetMeterValue(i) );       // Call our virtual GetMeterValue
                    }
          }
     }
}
```

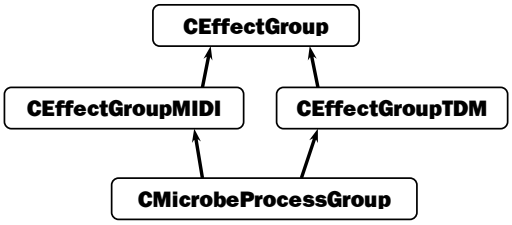
Notice the system call to `TickCount()`. This additional "tick tracking" code helps throttle the refresh rate of the meters to something reasonable. A *tick* in the MacOS system is approximately 1/60th of a second. In addition, this method has to call the platform dependent `GetMeterValue()` method which are implemented separately in the `CTemplateProcessAS` and `CTemplateProcessMuSh` classes.

Chapter 15: The Microbe Plug-In

The Microbe is an RTAS and Non-MuSh TDM plug-in which derives from the Effect Layer classes of the Plug-In Library. The Microbe is intended to form the basic framework of a plug-in synthesizer and implements some basic MIDI functionality. Unfortunately, the Microbe doesn't actually implement any audio synthesis algorithms - instead, just a basic volume control. Oh, well.

The Microbe Group Level

The Microbe Group level multiply inherits from two classes, `CEffectGroupTDM` and `CEffectGroupMIDI`. `CEffectGroupMIDI` "mixes-in" MIDI functionality to `CEffectGroupTDM`.



There are three main pieces composing the Microbe's Group level. They are the constructor, the `Initialize()` method, and the `CreateEffectTypes()` method.

Constructor

```
CMicrobeProcessGroup::CMicrobeProcessGroup(void)
{
     DefinePlugInNamesAndVersion("Microbe MIDI Plug-In\nMicrobeMIDI\nMicrobe", 1);
     DefineManufacturerNamesAndID("Digidesign\nDigi", 'Digi');

     AddGestalt(pluginGestalt_SupportsDeckChange);     // TDM & RTAS pi types are interchangeable
     AddGestalt(pluginGestalt_IsCacheable);            // Plug-in will be cached for faster DAE loading

     DefineDspResourceAndMaxChannels(2, kMerleType,    EffectLayerDef::ALL_CORE_TYPES, kMaxNumChannelsOnPCIHW);
     DefineDspResourceAndMaxChannels(3, kSatchmoType,  EffectLayerDef::ALL_CORE_TYPES, kMaxNumChannelsOnOnyxHW);
     DefineDspResourceAndMaxChannels(4, kGershwinType, EffectLayerDef::ALL_CORE_TYPES,
                                                       kMaxNumChannelsOnPrestoAt48k,
                                                       kMaxNumChannelsOnPrestoAt96k,
                                                       kMaxNumChannelsOnPrestoAt192k);
     DefineDspResourceAndMaxChannels(5, kTDM2GenericType,   EffectLayerDef::ALL_321_CORE_TYPES,
                                                            kMaxNumChannelsOn321At48k,
                                                            kMaxNumChannelsOn321At96k,
                                                            kMaxNumChannelsOn321At192k);
}
```

■ The constructor defines all the parameters of the overall plug-in. The first two calls should be self explanatory; they define the plug-in's name and version number. The version number is stored in save-and-restore settings files and is used to connect to SDS; otherwise, the Effect Layer makes little use of this information, but it's comforting knowing it's around. The second call defines your manufacturer name and your unique 32-bit identifier. Several names can be provided of varying string lengths, separated by the '`\n`' escape sequence. DAE and Pro Tools can then selectively choose one of the strings dependent on display size restrictions.

■ The second set of calls define a couple of properties, or gestalts, of your overall plug-in. The first *SupportsDeckChange* states that your Effect Types, RTAS and TDM, are interchangeable and will be swapped transparently if the playback engine of Pro Tools changes. This interchanging is handled by the Effect Layer. The *IsCachable* gestalt states that it's okay for Pro Tools to cache your plug-in the first time it's loaded. Refer to the section **Cachable Plug-Ins** in the chapter **General Topics** for more information on plug-in caching. Virtually all plug-ins should declare this gestalt.

■ The last "Defines" inform the Effect Layer what DSP code binaries exist in the plug-in, on what hardware they run on, and how many channels they support using the channel allocation system.

Initialize

```
void CMicrobeProcessGroup::Initialize (void)
{
     CEffectGroupTDM::Initialize ();    // Always call inherited first

     // add our custom control views to the system
     CCustomView::AddNewViewProc((StringPtr) "\022BackgroundPictView", CreateCBackgroundPictView);
     CBackgroundPictView::InitBackgroundPictView();

     CCustomView::AddNewViewProc((StringPtr) "\015HighlightView", CreateCHighlightControlEditor);
     CCustomView::AddNewViewProc((StringPtr) "\012PictButton", CreateCPictButton);
     CCustomView::AddNewViewProc((StringPtr) "\030PictBackgroundTextEditor", CreateCPictBackgroundTextEditor);
     CCustomView::AddNewViewProc((StringPtr) "\015GainMeterView", CreateCGainMeterView);
     CCustomView::AddKeyword((StringPtr) "\006labels",               kConstantElem,      CMeterView::kLabelKeyword);
     CCustomView::AddNewViewProc((StringPtr) "\026StateTextControlEditor", CreateCStateTextControlEditor);
     CCustomView::AddNewViewProc((StringPtr) "\011FrameView", CreateCFrameView);
     CCustomView::AddNewViewProc((StringPtr) "\017TextCustomPopup", CreateCTextCustomPopupEditor);

     CMultiPanelView::RegisterView();
     CSliderPictControlEditor::RegisterView();
     CMicrobeKeyboard::RegisterView();
     CPictDialControlEditor::RegisterView();
}
```

■ Always ensure that you call the inherited class method first; in this case it is `CEffectGroupTDM::Initialize()`.

■ The `Initialize` method is probably the ugliest thing that you need to implement. Essentially it's used to setup anything needed for the global plug-in. This is a good place to register all of your views with the GUI system. This way it is aware of them when it generates any GUI windows at the Process level. Ideally, all views should have a `static RegisterView()` method that can be called, like is seen with `CSliderPictControlEditor` or `CMicrobeKeyboard`; however, not every view class in the Plug-In Library has been updated with this method as of yet. Instead, you may perform the equivalent `AddNewViewProc` and `AddKeyword` calls.

■ Lastly, you must implement the pure virtual method `HandleDirectMidiEvent()` to handle application messages from the DirectMidi system. The Microbe's is trivial and looks like:

```
//======================================================================
//
//        METHOD: HandleDirectMidiEvent()
//        The main MIDI event handler
//
//======================================================================
void CMicrobeProcess::HandleDirectMidiEvent(DirectMidiCallbackType type, DirectMidiPacket* inPacket, Cmn_UInt32
nodeRefCon)
{
          CMicrobeProcess* myProcessPtr = reinterpret_cast<CMicrobeProcess *>(nodeRefCon);
          if (type == eMidiMessage)
          {
                    // MIDI Packet handling code goes here
                    myProcessPtr->MidiStream(inPacket->mData, inPacket->mLength);         //Parse MIDI stream.
                    //This will eventually call ParseNoteOn() if it's a NoteOn message.

                    //Except for Beat Clock and SPP, turn on our "MIDI IN" LED.
                    if(inPacket->mData[0]!=kMIDISystemRealtime && inPacket->mData[0]!=kMIDISongPositionPointer)
                              myProcessPtr->fMIDILightState=1.0 + myProcessPtr->fMIDILightState *
kMIDILightFastDecayRate;
          }
}
```

CreateEffectTypes

```
void CMicrobeProcessGroup::CreateEffectTypes (void)
{
     //Define the EffectType's ID, product ID, platform, and category -- really it's not a SW Generator
     CEffectType *microbeRTAS = new CEffectTypeRTAS('mmmc', 'MicW', ePlugInCategory_SWGenerators);
     // Type Names should listed add longest first to shortest last.
     microbeRTAS->DefineTypeNames("Microbe\nMicb\nMic");
     // Enable the bypass button
```

```
    microbeRTAS->AddGestalt(pluginGestalt_CanBypass);
    // Enable support for variable RTAS buffer sizes
    microbeRTAS->AddGestalt(pluginGestalt_SupportsVariableQuanta);
    // Enable support for a sidechain input (simply reroutes as the new input.)
    microbeRTAS->AddGestalt(pluginGestalt_SideChainInput);
    // Supports all sample rates.
    microbeRTAS->DefineSampleRateSupport(eSupports48kAnd96kAnd192k);
    // This is a mono in, mono out plug-in.
    microbeRTAS->DefineStemFormats(ePlugIn_StemFormat_Mono,ePlugIn_StemFormat_Mono);
    // Use the "MonoPageTables" set of page tables.
    // PgTL, FrTL, PcTL, and MkTL Resources have been defined in the MicrobePageTables.r
    microbeRTAS->DefinePageTableName("MonoPageTables");
    // Give this type life!  Allow it instantiate Processes.
    microbeRTAS->AttachEffectProcessCreator(NewMicrobeASPProcess);


    // Now let's make a TDM Type.
    CEffectType *microbeTDM = new CEffectTypeTDM('11Mc', 'MicW');
    // And copy the RTAS Type attributes to this TDM type.
    *microbeTDM = *microbeRTAS;
    // Define the sample rate support for Satchmo and Gershwin.
    microbeTDM->DefineSampleRateSupportForCardType(eSupports48kOnly,              kSatchmoType);
    microbeTDM->DefineSampleRateSupportForCardType(eSupports48kAnd96kOnly, kGershwinType);
    microbeTDM->DefineSampleRateSupportForCardType(eSupports48kAnd96kOnly, kTDM2GenericType);
    microbeTDM->AddGestalt(pluginGestalt_WantsTDMRunningTime);

    // Set the correct Process Creator.
    microbeTDM->AttachEffectProcessCreator(NewMicrobeTDMProcess);

    // Lastly, lets declare a stereo TDM Type...
    CEffectType *microbeTDM22 = new CEffectTypeTDM('22Mc', 'MicW');
    *microbeTDM22 = *microbeTDM;
    // Make sure it has stereo in, and stereo out for its ports.
    microbeTDM22->DefineStemFormats(ePlugIn_StemFormat_Stereo, ePlugIn_StemFormat_Stereo);
    // The Stereo Type has the same controls as the mono, so no need for new page tables.

    // Register our three types with the Group:
    AddEffectType(microbeRTAS);
    AddEffectType(microbeTDM);
    AddEffectType(microbeTDM22);
}
```

In this method, you must declare all the Types that your plug-in provides. In addition, you must define all the general attributes of these Types. This will be discussed briefly, but refer to the Appendices for the definitive guide on the Effect Layer.

Here, we define the three Types of the Microbe: A mono RTAS, and a mono and stereo TDM Type. The Effect Type's assignment operator has been implemented to aid in the initialization of each Type. Once the first Type's attributes have been defined they can be assigned to the others to reduce redundant code.

For example, `*microbeTDM = *microbeRTAS;` transfers the RTAS's name, gestalts, stem formats, plug-in category, page table name, and Effect Process creator function to the mono TDM Type. Subsequently, sidechain functionality is added the mono TDM Type, and its Effect Process Creator is redefined to the correct one.

Last, the mono TDM properties are assigned to the stereo one, and then its stem formats are adjusted to be stereo.

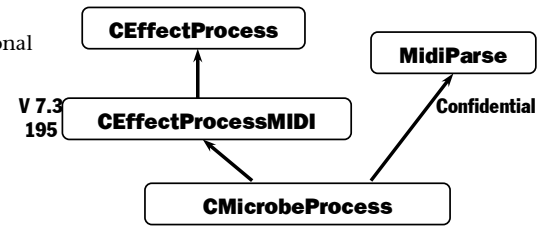
The `AttachEffectProcessCreator()` method gives the Type a pointer to a function which allows it to locally instantiate its corresponding Effect Processes. For example, the RTAS Effect Process Creator is simply the following:

```
CEffectProcess* NewMicrobeASPProcess()
{
     return new CMicrobeASPProcess;
}
```

Note that all Effect Types should be heap created (i.e. use `new`). Finally, the Type is registered with the Group via the `AddEffectType()` call.

The Microbe Processes

The main CMicrobeProcess level inherits from `CEffectProcessMIDI`, which provides additional



MIDI functionality. It also inherits from the helper class `MidiParse` to help read the incoming MIDI streams.

(The Microbe example implements a single Process for both the mono and stereo versions, performing runtime switching, using the `GetNumOutputs()` call. This may be appropriate for such a simplistic plug-in to reduce class clutter; however, for something more complex it might be more suitable and efficient to implement these via separate classes.)

`CMicrobeProcess` implements all functionality common to both the RTAS and TDM Types. There are several core functions of this class. They are to:

1 Initialize the Microbe's controls and add them to the Control Manager. This is performed in the constructor of the class.
2 Define the DLOG resource source for generating the GUI, along with initializing view elements: `EffectInit()`, `SetViewPort()`.
3 Update the views of the GUI during run-time: `DoTokenIdle()`, `UpdateIdleGraphics()`, `UpdateControlGraphic()`, `UpdateMeters()`.
4 Handle key shortcuts from the user: `HandleKeystroke()`.
5 Parse the incoming MIDI stream: `HandleDirectMidiEvent()`, `ParseNoteOn()`, etc.

We will now look at some of this functionality in more detail.

1 First off, the constructor:

```
CMicrobeProcess::CMicrobeProcess(void)
: fMultiPanelSelectorIndex(0), fMIDILightState(0.0), mLastMeterTicks(0)
// Initialize the multi-panel to the first panel, initialize the state of the midi indicator light, and initialize
// the Tick counter that is used to throttle the rate of graphic refreshes.
{
     // Now, add our controls to the control manager:

     // kMasterbypass = 1:
     //-----------------------------------------------------------------------------------------------------------------
     // default = off
     // automatable = true
     AddControl(new CPluginControl_OnOff('bypa', "Master Bypass\nMstr Byp\nMBYP\nByp", false, true)); // Default to off
     DefineMasterBypassControlIndex(kMasterbypass);

     // kGain = 2:
     //-----------------------------------------------------------------------------------------------------------------
     // min = 0.0 (-inf dB)
     // max = 1.0 (0 dB)
     // step = 0.2
     // default = 1.0 (0dB)
     // automatable = true
     //
     // The 0.2 step size will give 6 steps between -infinity and 0 dB.  This will be useful for the initialization
     // of the radio button style volume selector, which has 6 buttons, and thus the text popup will also have 6
     // selections.  In contrast, the slider will still act as a continuous control, and won't be quantized to six
     // steps.  However, note that under standard mode on Procontrol it will be quantized to the these 6 steps.
     AddControl(new CPluginControl_LinearGain('ganL', "Gain\nGan", 0.0, 1.0, 0.2, 1.0, true));

     // kMultiPanelSelectCtrl = 3:
     //-----------------------------------------------------------------------------------------------------------------
     // min = 0
     // max = 3
     // default = 0
     // automatable = false
     //
     // We don't enable automation for the radio button multipanel selector cause this would be weird.
     AddControl(new CPluginControl_Discrete(' RG1', "MultiPanel Selector\nMPl Sel\nMPSl\nMPS", 0, 3, 0, false));
     // Disable this from being save-and-restored since it really isn't a parameter.
     FilterControlIdOnSave(' RG1');
}
```

The constructor should be well explained by its comments alone. Refer to the chapter on GUI development for more reference.

2 By the time `EffectInit()` is invoked, the Process is "aware" of who it is, what Type is belongs to, how many inputs and outputs it has, etc. This is the place to make any final initializations before the Process starts that might be dynamically dependent on this information. In the Microbe, we use this method to define the GUI DLOG resource based upon whether this is a mono or stereo Process. The GUIs differ in the number of meters they have.

```
void CMicrobeProcess::EffectInit(void)
{
     // Let's define our what DLOG our GUI is coming from:
     if (GetNumOutputs() == 2)
          this->DefineDLOGId(2);        // Stereo DLOG is in ID# 2
     else
          this->DefineDLOGId(128);      // Mono is ID# 128
}
```

`SetViewPort()` is called any time the Process is being attached to, or detached from, the GUI window. By overriding this method we have the opportunity to "hook-up" and initialize any view elements our Process code might use. In the Microbe, we use this method to find several views so that they can be updated during processing time. These include meters, the keyboard view, the MIDI indicator light, and a multipanel view.

```
void CMicrobeProcess::SetViewPort (GrafPtr aPort)
{

     // Always start with inherited, in order to attach control to view and init view.
     CEffectProcess::SetViewPort(aPort);

     // Bail if we couldn't create a plug-in view.
     if (fOurPlugInView == nil)
          return;
     // Get a pointers to the  meter objects so that we can update them at idle time.
     mGainMeterView[kLeftChannel] = dynamic_cast<CGainMeterView *>(fOurPlugInView->FindSubView(MeterViewIDLeft));

     mGainMeterView[kRightChannel] = dynamic_cast<CGainMeterView *>(fOurPlugInView->FindSubView(MeterViewIDRight));

     // Get a pointer to the keyboard object so that we can update it at idle time.
     mMouseKeyboardView = dynamic_cast<CMicrobeKeyboard *>(fOurPlugInView->FindSubView(MouseKeyboardViewID));
     if (mMouseKeyboardView)
     {
          mMouseKeyboardView->SetOwner(this);
          mMouseKeyboardView->SetMaxMinNote(kMaxMouseKeyboardNote, kMinMouseKeyboardNote);
     }

     mMultiPanelView = dynamic_cast<CMultiPanelView *>(fOurPlugInView->FindSubView(MultiPanelViewID));
     if(mMultiPanelView)
          mMultiPanelView->SetCurrentPanel(fMultiPanelSelectorIndex);

     fMIDILEDView = dynamic_cast<CPictDialControlEditor *>(fOurPlugInView->FindSubView(MIDILEDViewID));
     if(fMIDILEDView)
     {
          fMIDILEDView->SetValue(EffectLayerDef::OFF_CONTROL_VALUE);
          fMIDILEDView->RedrawIfChanged();
     }
}
```

3 Now, during the running of the Process, these view elements need to periodically updated for the user. This updating originates in the overridden method `DoTokenIdle()`. First we check that a significant enough time span has occurred since the last update, via the system `TickCount()`. If so, we go ahead and do the updating. MacOS *ticks* are roughly equivalent to 1/60th of a second.

```
void CMicrobeProcess::DoTokenIdle (void)
{
     CEffectProcess::DoTokenIdle();

     long Ticks = TickCount();
     if (Ticks != mLastMeterTicks) {
          mLastMeterTicks = Ticks;

          this->UpdateMeters();         // update view meters during idle time
          this->UpdateIdleGraphics();   //update MIDI led, etc.
     }
}
```

In `UpdateMeters()`, the value of the meter is requested from the algorithm using the `GetMeterValue()` method. Since at the CMicrobeProcess level, the processing algorithm hasn't yet been implemented, we force `GetMeterValue()` to be a pure virtual function that must be implemented in the inherited RTAS and TDM classes. `UpdateMeters()` is shown here; refer to the source code for the implementation of `UpdateIdleGraphics()`.

```
void CMicrobeProcess::UpdateMeters(void)
{
     // update meter views with the current values
     if (mGainMeterView[kLeftChannel])
          mGainMeterView[kLeftChannel]->SetValue(this->GetMeterValue(kLeftChannel+1));
     if (mGainMeterView[kRightChannel])
          mGainMeterView[kRightChannel]->SetValue(this->GetMeterValue(kRightChannel+1));
```

```
}
```

In the Microbe, the Plug-in Library call `UpdateControlGraphic()` is overridden for special functionality. This method is invoked anytime view graphics need to be updated in response to the control changing either via the user input or automation. Here, we catch updates to our group of radio buttons and transfer this update to the multipanel display which these buttons control.

```
ComponentResult CMicrobeProcess::UpdateControlGraphic (long aControlIndex, long aValue)
{
     // We need to intercept changes to the multipanel radio buttons to update the multipanel.
     if(aControlIndex == kMultiPanelSelectCtrl)
     {
          if(mMultiPanelView)
               mMultiPanelView->SetCurrentPanel(fMultiPanelSelectorIndex);
     }

     return CEffectProcess::UpdateControlGraphic(aControlIndex, aValue);
}
```

4 User input via the keyboard is also handled at the `CMicrobeProcess` level. Here we catch *ctrl*+← and *ctrl*+→ keystrokes and alter the gain control up and down. To do this we query the Control Manager for the continuous (`double`) value of the `kGain` control in the line:

```
double gain = dynamic_cast<CPluginControl_Continuous*>(GetControl(kGain))->GetContinuous();
```

After the gain is manipulated, it is sent back to the control:

```
controlValue = dynamic_cast<CPluginControl_Continuous*>(GetControl(kGain))->ConvertContinuousToControl(gain);
SetControlValue(kGain, controlValue);
```

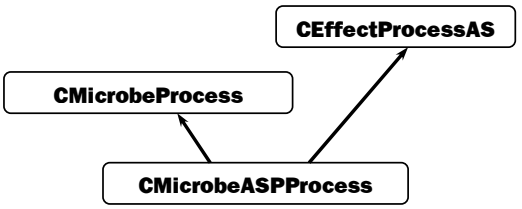
`SetControlValue()` expects DAE's signed 32-bit integer format, therefore the conversion from a double must happen using the control's helper method `ConvertContinuousToControl()`. Following is the complete `HandleKeystroke()` method. This method returns *true* if the key event was handled. This alerts DAE that it shouldn't attempt to pass the event on to Pro Tools.

```
bool CMicrobeProcess::HandleKeystroke(EventRecord *theEvent)
{
     bool usedEvent = false;
     long controlValue;

     // Using the CTRL-key so that we do not conflict with the DAE-Aware Application
     // Key Commands.
     if (theEvent->modifiers & controlKey)
     {
          if ((theEvent->what == keyDown) || (theEvent->what == autoKey))
          {
               double gain = dynamic_cast<CPluginControl_Continuous*>(GetControl(kGain))->GetContinuous();

               // Right arrow key for Gain Increase
               if ((theEvent->message & charCodeMask) == 0x1D)
               {
                    gain += 0.02;
                    if (gain > 1.0) gain = 1.0;
                    usedEvent = true;
               }

               // Left arrow key for Left Port Gain Decrease
               if ((theEvent->message & charCodeMask) == 0x1C)
               {
                    gain -= 0.02;
                    if (gain < 0.0) gain = 0.0;
                    usedEvent = true;
               }

               if (usedEvent == true) {
                    controlValue =
                         dynamic_cast<CPluginControl_Continuous*>(GetControl(kGain))->ConvertContinuousToControl(gain);
                    SetControlValue(kGain, controlValue);
               }
          }
     }

     return usedEvent;
}
```

5 Lastly, `HandleDirectMidiEvent()` is implemented to process the incoming MIDI stream. Functionality from the `MidiParse` class is used to extract out the individual MIDI messages. Refer to the source code for more detail.

RTAS Microbe Process

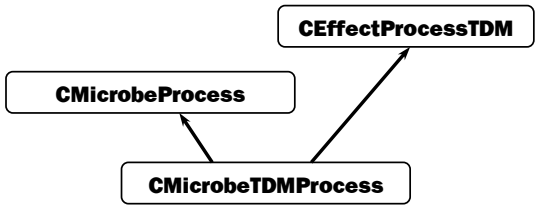
The RTAS Process class `CMicrobeASPProcess` implements the remaining RTAS specific functionality. It does this by multiple inheritance, and mixing-in `CEffectProcessAS`, which provides some underlying AS and RTAS services to the base class. Note that `CMicrobeProcess` and `CEffectProcessAS` are both `virtual CEffectProcess`es to prevent the dreaded diamond shape from occurring in the inheritance tree.



In the RTAS Process class, `ProcessAudio()`, `UpdateControlValueInAlgorithm()`, and `GetMeterValue()` are implemented. `ProcessAudio()` accepts incoming audio buffers and performs the gain "algorithm." `UpdateControlValueInAlgorithm()` updates the parameters of the algorithm based upon changes in the plug-ins controls, and lastly, `GetMeterValue()` extracts out a meter value from the algorithm so that the lower class `CMicrobeProcess` can use it to update the visual gain meters.

TDM Microbe Process

`CMicrobeTDMProcess` fulfills the same requirements as the RTAS Process class. It updates the algorithm, which is now running on the TDM system; hence, there is no `ProcessAudio()` method. It also implements the `GetMeterValue()` to retrieve meter values from the TDM DSP code.



Additionally, the `SetDSPInfo()` method is overridden to setup the state of the DSP code when it first begins processing. Lastly, `ConnectSidechainTDM()` and `DisconnectSidechainTDM()` are overridden to handle the details of connecting a sidechain input to the DSP Process.

Host Commands

The Microbe utilizes four host commands for DSP communication. (These are in addition to other underlying host commands that the plug-in Library and Effect Layer may initiate.) They are named `hcGain`, `hcGetOutputMeter`, `hcSetBypass`, and `hcSidechain`, and utilize Host Command Vectors 2 through 5 of the 56K DSP.

Let us look at how one of the host commands is executed in the `UpdateControlValueInAlgorithm() method()`.

```
void CMicrobeTDMProcess::UpdateControlValueInAlgorithm (long controlIndex)
{
     double gain;
     SInt32 dspGain;
     UInt32 bypass;

     switch (controlIndex)
     {
          case kMasterbypass:
               bypass = dynamic_cast<CPluginControl_Discrete *>(GetControl(controlIndex))->GetDiscrete();
               // This discrete control returns either 0 or 1.
               bypass *= 0xFFFFFF;
               if(GetNumOutputs() == 1)
                    SendControlValueToDSP(1, hcSetBypass, bypass);
               else {
                    SendControlValueToDSP(1, hcSetBypass, bypass);
                    SendControlValueToDSP(2, hcSetBypass, bypass);
               }
```

```
            break;

        case kGain:
            gain = dynamic_cast<CPluginControl_Continuous*>(GetControl(controlIndex))->GetContinuous();
            dspGain = gain * double(kMaxDSPGain);

            if(GetNumOutputs() == 1)
                SendControlValueToDSP(1, hcGain, dspGain);
            else {
                SendControlValueToDSP(1, hcGain, dspGain);
                SendControlValueToDSP(2, hcGain, dspGain);
            }
            break;

        case kMultiPanelSelectCtrl:
            fMultiPanelSelectorIndex
                = dynamic_cast<CPluginControl_Discrete *>(GetControl(controlIndex))->GetDiscrete();
            break;
    }
}
```

In particular, look at the handling of the `kGain` control. Its double value is extracted from the control and converted an appropriate 24-bit DSP fixed point value. This value is then sent to the DSP for port 1, and port 2 if this is a stereo Process, using the `SendControlValueToDSP()` call.

The TDM DSP Code

The Microbe's DSP assembly code is found in the file `MicrobeProcess.asm`.

Interrupt Vector Table

```
    ORG  P:ResetDSP
    JMP  >ProgramStart

    ORG  P:TDMInterrupt ; TDM interrupt
    JSR  <DoTDMInt

    ORG  P:HostCmdInt0  ; Called by standard DSP TDM object
    JSR  <hcInitialize

    ORG  P:HostCmdInt1  ; Called by standard DSP TDM object
    JSR  <hcStartDSP

    ORG  P:HostCmdInt2  ; Get "Gain"
    JSR  <hcGain

    ORG  P:HostCmdInt3  ; Get Output Meter
    JSR  <hcGetOutputMeter

    ORG  P:HostCmdInt4  ; Get Master Bypass value
    JSR  <hcSetBypass

    ORG  P:HostCmdInt5  ; Get Sidechain Key value
    JSR  <hcSidechain

    ORG  P:HostCmdInt6  ; Get channel offset
    JSR  <hcSetChannelOffset

    ORG  P:HostCmdInt13 ; All TDM config. commands filter thru here.
    JSR  <hcTDMCommand
```

Chapter 16:
The Surround Downmixer Plug-In

The Surround Downmixer plug-in is a TDM2-only plug-in. It uses the DSP's DMA controller to do its audio I/O with the TDM2 ASIC. Additionally, this plug-in demonstrates an advanced feature of the TDM2 ASIC: the Re-ordering Tables (ROTs). By using DMA and the ROTs, a plug-in can achieve better cycle efficiency and therefore, perhaps, increased instance counts. As a *very* rough estimate, using DMA and the ROTs is advantageous if your plug-in can achieve around 15 or more mono-to-mono instances at 44.1kHz using programmed I/O. In this case, a couple more instances may be possible if using DMA. With higher instance counts, the gains are much greater, since the number of cycles saved by removing programmed I/O approaches the number cycles actually required to process that I/O.

The Surround Downmixer contains two plug-in Types: a 5.1-to-stereo Type and a LCRS-to-stereo Type. The algorithms for both Types is very simple. Two gain, or rather attenuation, values are multiplied separately on each input and then summed to create left and right stereo outputs. The multipliers are set such that left input channels are panned hard left, center channels are mixed halfway between left and right, etc. The host side code adjusts these gain values to either mute or unmute the input channels individually. These gain values are stored in parameter blocks on the DSP. Since 2 values are needed for each input channel, the 5.1 Type has a parameter blocks size of 12 words and the LCRS a size of 8 words.

Examining the DSP Code

Let's start by looking at the TDM interrupt. Its basic steps are shown below. To do the transfer of input and output samples, a double buffering scheme is used. Step 2 deals with switching the pointers to these buffers and triggering DMA to operate on the opposing set of buffers.

1 Save registers in internal memory.

2 Switch the double buffers.
- **A** Change the DMA controller source and destination registers to point to the other set of input and output buffers.
- **B** Change the `r4` and `r5` pointers to point to the opposite set of input and output buffers.

3 Trigger the DMA sequence to start.

`r4` and `r5` are then used throughout the rest of the TDM interrupt, while the DMA controller "silently" reads in the newest set of samples, and writes the output samples that were processed in the previous TDM interrupt.

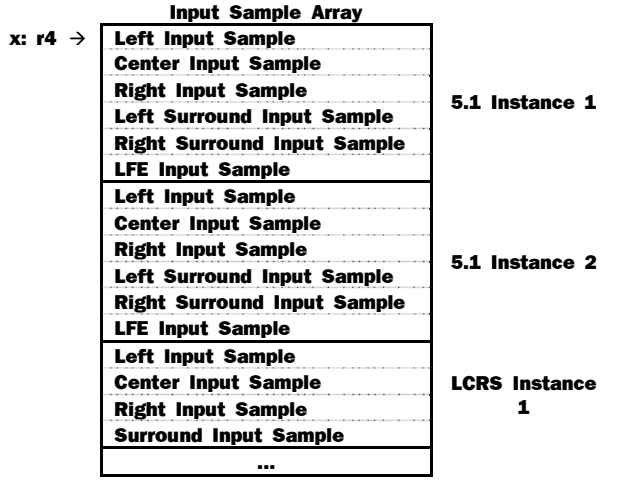
4 Process the 5.1 instances, followed by the LCRS instances.

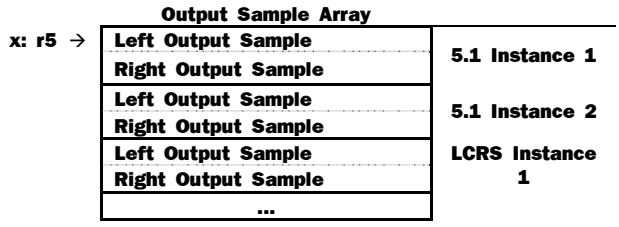
5 Restore saved registers.

DSP Data Structures

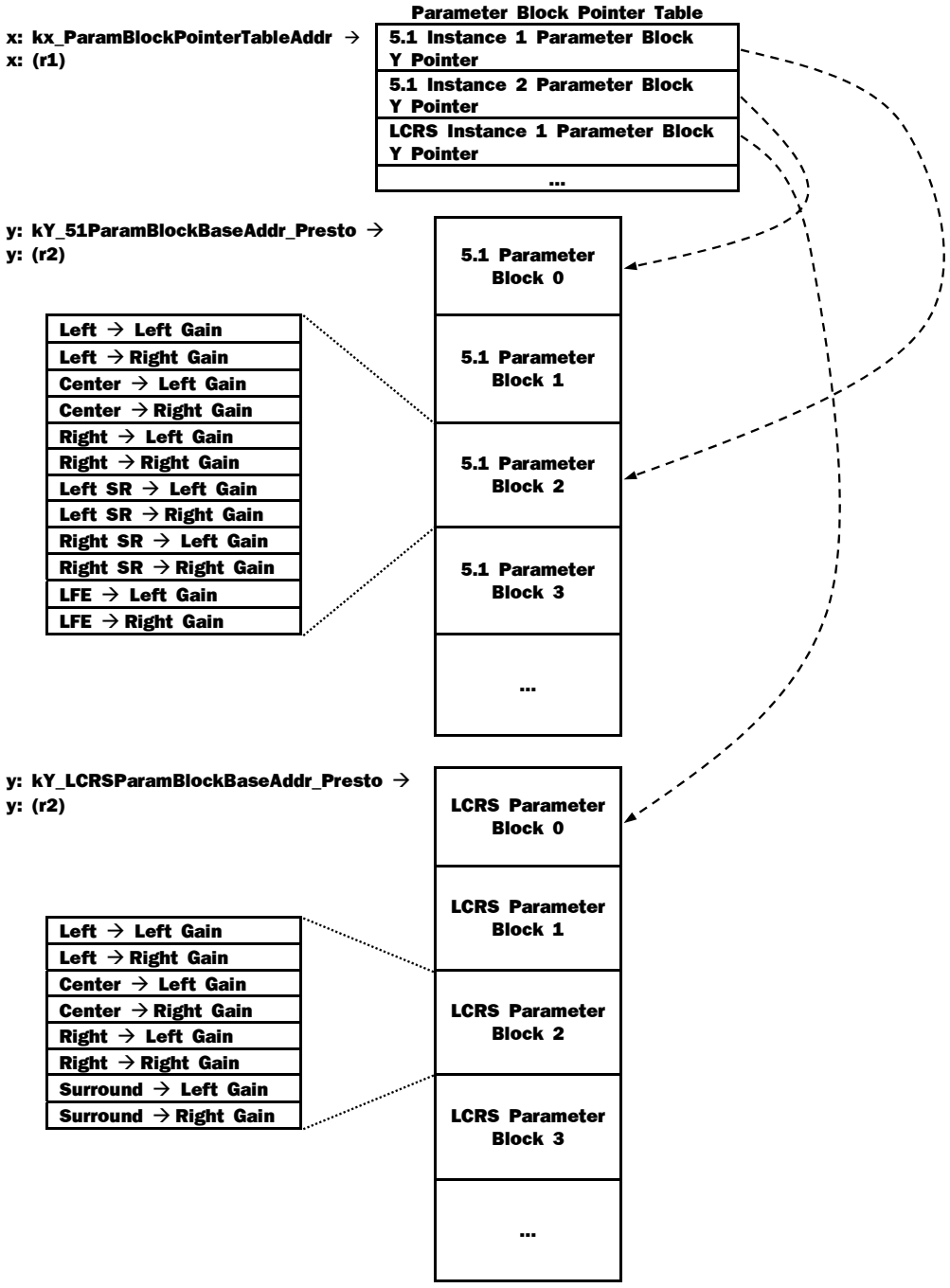
There are seven primary data structures on the DSP. The double "Input Sample Arrays," the double "Output Sample Arrays," the "Parameter Block Pointer Table," the set of "5.1 Parameter Blocks," and the set of "LCRS Parameter Blocks."

Below, a conceptual diagram of a single set of input and output sample arrays is shown. Through programming the TDM2's Reordering Tables , the input and output sample arrays have been logically and consecutively arranged. This greatly simplifies the DSP code of the inner processing loops. The functionality of the ROT's is examined later in the host code discussion.





The next page illustrates the Parameter Block Pointer Table and the Parameter Blocks themselves. This indirection between the instances themselves and their associated Parameter Blocks makes reorganizing the instances easier from the host-side code. For example, if an instance is removed, only the pointer in Pointer Table needs to removed, and the table compacted. This is simpler than compacting the entire set of Parameter Blocks. This does however require some extra dereferencing at processing time to acquire the actual pointer to each instance's Parameter Block.



(Additionally shown are the host C++ constants and the DSP address registers that point to these structures. The hypothetical pointer arrows demonstrate how the pointer table references the parameter blocks.)

DMA Bug in the Presto DSP

The Presto DSP chip on the HD cards contains a few bugs related to DMA. The most critical one, involves directly enabling a DMA channel to transfer a block of data. Such DMA transfers might not always work properly. To circumvent this problem, the actual DMA transfers are triggered indirectly by the completion of a "dummy" DMA transfer. This dummy transfer *is* triggered directly by software. This dummy transfer might sometimes fail; however, it always successfully triggers any DMA channels that are chained to it.

In the sample plug-in's DSP code, channel 2 is used as the dummy DMA channel. Channel 3 is the real DMA channel that writes samples to the TDM ASIC and channel 4 is the real DMA channel that reads samples from the TDM ASIC. Channel 3's `DRS` bits (DMA Request Source Bits) are set to `00110` so that it is triggered by the completion of DMA channel 2. Channel 4's `DRS` bits are set to `00111` so that it is triggered by the complete of DMA channel 3. So, by triggering channel 2 all TDM read and writes are ensured to occur. The `DRS` bits are found in each DMA channel's `DCRx` register.

Channel 2's Source Address Register (`DSR2`), Destination Address Register (`DDR2`), and Counter (`DCO2`) are initialized in `hcStartDSP`. The dummy trigger is done in the `StartDMA` macro by writing `kDMA_Trigger` to channel 2's Control Register (`DCR2`). The `kDMA_Trigger` bit pattern can be found in the sample DSP code.

For brevity's sake, the DSP code in the example plug-in is the same for both 321 and Presto chips. Although they share the same DSP code, the 321 does not exhibit this bug and does not actually need these "dummy" DMA transfers.

The TDM2 Reordering Tables (ROTs)

The Reordering Tables are lookup tables that remap DMA read and writes with any arbitrary input or output TDM timeslot. In short, the ROTs allow you to organize the sequence of all inputs and outputs to your algorithms however you please.

To program the ROTs, several DSI API's are used. They include:

```
Dhm_TDM2_PrepareForConnections(SInt32 iPrepareBool)
Dhm_TDM2_MapReads(SInt32 iNode, SInt32 iROTAddr, SInt32 iRAMAddr);
Dhm_TDM2_MapWrites(SInt32 iNode, SInt32 iROTAddr, SInt32 iRAMAddr);
Dhm_TDM2_GetNodeNumber(struct DSPRecord *, SInt32 *oNodeNum);
```

To begin reprogramming the ROT tables, `Dhm_TDM2_PrepareForConnections(true)` must be called. To terminate and commit a sequence of remappings, `Dhm_TDM2_PrepareForConnections(false)` must be called.

`GetNodeNumber()` acquires the TDM2 node index of your DSP's particular TDM2 ASIC. The node number is then specified to remap that node's ROT tables in `Dhm_TDM2_MapWrites()` and `Dhm_TDM2_MapReads()`.

`Dhm_TDM2_MapReads()` modifies the Read ROT and thus affects the ordering of incoming audio samples. Likewise `Dhm_TDM2_MapReads()` modifies the Write ROT and affects the ordering of outgoing audio samples. Both functions have the same arguments. iRAMAddr is, in essence, the TDM timeslot number and will have a value of 0 to 511. This value is provided by DAE in the 2nd argument of the `ConnectInput()` and `ConnectOutput()` routines, which is usually defined as "`long cardChannel`" in most Plug-In Library code. `iROTAddr` is the entry number in the ROT and should also have a range of 0 to 511. Last, when DMA is performed, the ASIC references the ROT table in sequence, starting at entry 0, to determine what timeslot it should read or write from. So, by writing the `iRAMAddr` (timeslot) value of an associated port number (See the MultichannelPlugInSpec.doc for the standard channel ordering) into an arbitrary ROTAddr location, it is possible to create any sequence of input or output samples as desired. Additionally, it's even possible to map a single input to several locations in the sequence.

The internal logic of the ASIC to handle the Reordering Tables is pipelined. This requires that extra dummy reads and writes are done during the DMA sequence to fully flush out these pipelines. When doing DMA reads from the ASIC 7 extra reads are required. 5 extra writes are also required. This means that the first 7 words written to RAM by the DMA controller are invalid and should be ignored. In the sample DSP code this occurs in `SwapDoubleBuffers`, when `kNumTDM2DummyReads` is added to base address of the input buffers before placing the value in register `r4`.

The Host Code

The Process level call `SetHardwareRefNum()` is invoked any time the instance is added to or removed from a DSP. In `CSurroundDownmixerProcessTDM`, this opportunity is used to assign or release a Parameter Block on the DSP for that instance. The Group level Effect Layer calls, `AddChannelUser()` and `RemoveChannel()`, are used to acquire and release an index number that is later used to reference a particular Parameter Block. `AddChannelUser()` returns the lowest un-acquired index number and links that index number with a reference value that's passed in. (See the sample code for more details.)

Additionally, `SetDSPInfo()` is also called when the instance is added or removed to a new DSP. In `CSurroundDownmixerProcessTDM`, this function is overridden and used to "start" and "stop" the instance running on the DSP. These processes are fairly complicated. (See the sample code and accompanied comments for more information.) To rearrange the state of the DSP instances, the TDM interrupt is temporarily disabled. Since audio processing is halted momentarily, a small click in the audio might be heard.

Whenever the ROTs are manipulated, either in `StartInstanceOnDsp()`, `StopInstanceOnDsp()`, or any of the I/O connection routines (e.g., `ConnectInput()`), this state information is logged at the Group level, via the `LogRotMapping()` function. ROT state is retrieved with the `GetRotMapping()` function. This is required because any instance (Process) is capable of remapping the ROTs in their `StartInstanceOnDsp()` or `StopInstanceOnDsp()` functions. In order to shift entries in the ROTs, their entire state must be acquirable at the Process level.

Chapter 17: The Template_NoUI Plug-In

Introduction

The purpose of this plug-in is to demonstrate how to create a plug-in using a UI created with a framework other than the Digidesign View classes. Creating such a plug-in is more difficult on Windows given the fact that the plug-in library uses the Mac2Win Macintosh emulation library. Therefore, some of the architecture used to create this plug-in is specifically designed to get around this problem and would not be necessary if you were creating a Macintosh-only plug-in.

> *Note:* The plug-in has been completely updated in the 6.7 SDK. Therefore, while some of the documentation below still reflects how it worked in previous versions of the SDK, this documentation is specific to the 6.7 version and higher. The Mac version of the plug-in first appears in the final 6.7 SDK.

The strategy of this plug-in is to create a barrier between the plug-in's processing code and it's UI code. As an example, this plug-in uses the VST GUI library to create the UI. However, it is only intended as an example. You are free to use whichever framework you choose. However, in order to make clear which code is specific to using the VST GUI library, and which is not, the source code contains two categories of comments:

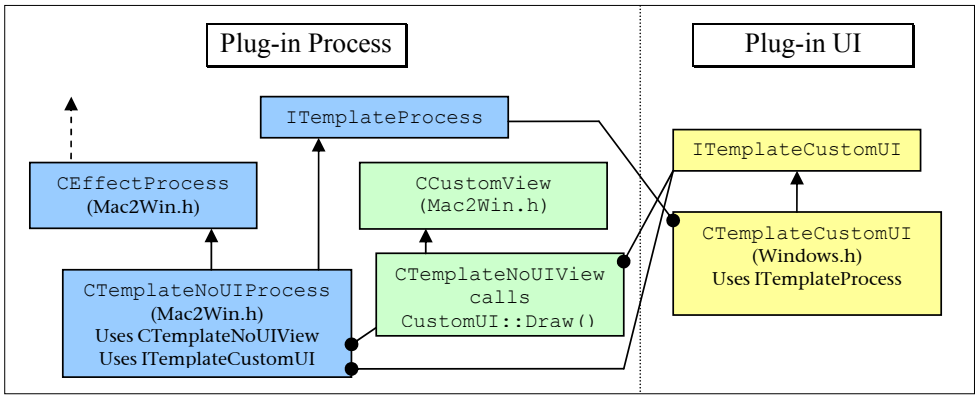
1) Comments relating to code required by any plug-in following the NoUI method; labeled with "NO_UI: ".
2) Comments relating to code specific to using the VST GUI library; labeled with "VSTGUI: ".

Searching for these two labels in the source files will allow you to easily find these comments. These comments are present in order to make it more clear when your plug-in uses another framework which parts will be necessary for a plug-in using this architecture (NO_UI) while making you aware of some of the issues that may come up in your framework (VSTGUI). However, keep in mind that Developer Services will not be able to help with framework-specific questions or problems. You will need to be responsible for being familiar with the framework you choose, how it works, and its limitations.

Architecture

The first step is to create an architecture that separates the plug-in's process from its UI, while allowing the two parts to communicate with each other. It may not be necessary to do this in all cases, i.e. if you're creating a Mac-only plug-in, but it is very necessary for a plug-in that will include any of the Windows headers. It is necessary because the Mac2Win.h header redefines many of the basic Windows types – like `HWND`. Therefore, if the Mac2Win.h header and the Windows.h header (or MFC headers) are included by the same file – directly or indirectly – there will be many compile-time name collisions.

In the example plug-in, this is accomplished by creating two interface classes, one for the process (ITemplateProcess), and one for the UI (ITemplateCustomUI). These interface classes should contain the declarations for any functions that must be called across the Process/UI boundary. Their declaration files should not contain include files that cause either Mac2Win.h or windows.h to be included. The figure below shows how the class structure works. The blue boxes represent the process-related classes and the yellow boxes represent the UI-related classes. The green boxes are somewhat of a hybrid in that they are technically part of the UI. However, they are part of the Digidesign View classes, use Mac2Win, and are therefore on the "Process" side. Perhaps a better name for the two sides would be Mac2Win and Windows rather than Process and UI, respectively.



Notice that any calls made across the Process/UI boundary are accomplished using the interface classes. Any methods in the Process class to be used by the UI (or vice versa) should be declared as virtual functions in the base interface class. As such, all function signatures should be platform independent. For example, instead of a function returning a Mac-specific OSErr, it should return a long.

The CTemplateNoUIView class overrides the DrawContents() and MouseCommand() (on Mac only) methods. The purpose of this class is so that the application can tell the plug-in window when it should draw itself, and what the proper clipping region should be. On the Mac, it also passes mouse events to the VSTGUI framework.

Implementation

The details for implementation will vary depending on your plug-in and the UI framework you decide to use. Therefore, the discussion below will only highlight the parts of the Template_NoUI plug-in that are generalizable to any plug-in employing this method.

Implementation: Plug-In Process (CTemplateNoUIProcess)

- The first step is to tell Pro Tools how big the plug-in window should be. Pro Tools will call `GetViewRect()` in the process level. Simply fill in the `Rect` to tell Pro Tools the size, in pixels, of the window it should create for the plug-in. In this example, it calls into the ITemplateCustomUI class to get this information.

- Once Pro Tools has the size information, it will call `CreateCPlugInView()` and create the window. The `CreateCPlugInView()` method should do whatever initialization is required by the UI, including creating an instance of the View class (CTemplateNoUIView, in this example). In this example, you'll see that the CTemplateNoUIView class is passed the pointer to the ITemplateCustomUI class so that it may call into that class.

- At this point, Pro Tools will call `SetViewPort()`. Pro Tools passes in the `GrafPtr` to the plug-in window as an argument. On Macintosh, you can simply use this pointer to begin drawing. On Windows, you'll need to call the Mac2Win function `ASI_GethWnd()` to obtain the `HWND` for the window. Pro Tools calls `SetViewPort()` both when opening and closing the window. The `GrafPtr` will be 0 when Pro Tools wants the plug-in to close the window.

  The size of the plug-in window created by Pro Tools is always larger in height and sometimes larger in width than the dimensions given in `GetViewRect()`. The reason is that Pro Tools adds the standard plug-in "header" at the top of the window, whose smallest width can sometimes be wider than the plug-in's width. On Windows, the `HWND` passed back from the call to `ASI_GethWnd()` is for the entire plug-in window, including the header. Therefore, the plug-in's UI must calculate the placement of its window below the header and centered. On the Mac version, this is not necessary,

since the port passed in from Pro Tools is already positioned properly below the header. See `CTemplateCustomUI::Open()` for further details.

• Override the process `SetControlValue()`. This method simply calls the inherited version, but overriding it here allows the method to be used by the UI. Keep in mind that it can be called by DAE when a control surface, a preset, or automation changes a control.

• Override the process level `UpdateControlGraphic()`. Calling `SetControlValue()` generates a token in the SDS system. Once it has been processed, DAE will call `UpdateControlGraphic()` to notify the plug-in that it should update the control. This is where the process should notify the UI to update the graphic.

• Override the process level `SetControlHighliteInfo()`. This function is called when a control is to be highlighted. The UI should be notified to change the control's highlight color. Since the Master Bypass control is on the plug-in header, the inherited CProcess function should be called for the bypass.

• Override the process level `ChooseControl()`. This function is called to find out the index of the control that contains the point (in local coordinates) passed in. This function is cross-platform, but is mainly overridden so that a Mac plug-in will have the default behavior when a user clicks on a control while holding down all three of the modifier keys: Cmd+option+control. See the discussion on modifier keys below for further information.

• Create other functions to be called by the UI in order to either set/get information in the process, or to tell the process to perform certain tasks. For example, the `Touch/ReleaseControl()` methods must be called when a mouse down/up event occurs (respectively) in order for Touch automation to work properly. The `ProcessDoIdle()` method is called by the UI when it has time for idle processes (like during a mouse down event) giving the rest of the plug-in – and the application in general – time to perform idle tasks. See `CTemplateNoUIProcess.cpp` for all the functions used here and read their comments to see their purpose. The functions here are by no means a complete list. Your plug-in may require other functions in order to allow other types of communication between the UI and the process.

Implementation: UI (CTemplateNoUIView)

• Derive a View class from `CCustomView` and override the `DrawContents()` method. The DrawContents method should call the Custom UI's Draw method, passing in the rectangle to be drawn. The presence of this view class allows the plug-in's custom UI to be drawn at the proper times and with a valid clipping rectangle.

Because of the differences between the Macintosh and Windows window systems, and since the Mac2Win layer emulates the Mac, not implementing this class will cause the plug-in's custom UI to be drawn at the wrong time – often with the background drawn on top of it. In addition, without this class, when the Custom UI's window receives WM_PAINT messages from Windows, the clipping rectangle will be incorrect, which would force the plug-in to draw the entire UI each time a WM_PAINT message is received. On the Mac, doing this allows the Custom UI to also draw at the correct times.

• On the Mac, override the `MouseCommand()` method. This was only required due to the VSTGUI library. Doing this may not be necessary for all frameworks.

Implementation: UI (CTemplateCustomUI)

• `Constructor`: For Windows plug-ins, if you are getting resources from the binary, you'll need the `HINSTANCE` on Windows or the FSSpec on Mac. It is stored in the global `gThisModule` variable, which is retrieved from the process in this example via the `ProcessGetModuleHandle()` method. The constructor in this example does two important things: It creates the UI objects and calls the `Init()` method.

• `Init()` window method. The `Init` method registers the window class with Windows from which the Windows-based window will be created in the `Open()` method. It does nothing on the Mac version.

• `Open()` window method. On Windows, the `Open()` method calculates the window position and then creates the window. This begins the message pump. For both the Mac and Windows versions, the VSTGUI-based window is created here as well. Keep in mind that plug-in windows are opened and closed frequently in Pro Tools so initialization tasks are better done in the constructor or in `Init()`, and not here.

• `Draw()` window method. On Windows, the `Draw` method simply calls the Windows API `InvalidateRect`, with the rectangle passed to it. On Mac, the UI's draw method is called. This is the method called by the CTemplateNoUIView class.

• On Windows, implement the callback for your window. `TemplateMainWindow()` is this plug-in's callback. In this example, since the VST GUI library has its own window procedure, this callback simply calls the default window procedure. If you are performing your own drawing, you can use this callback.

IMPORTANT: Whichever platform your UI is operating on, it should always allow any unprocessed messages to fall through to the parent window. This allows Pro Tools to process any commands that it should handle, like pressing the space bar to start / stop the Transport.

• Implement `UpdateGraphicControl()`. This is called by the process, telling the UI to update itself. This is the proper place for the UI to be updated. It is important that if the user moves a control the UI is NOT updated anywhere but in `UpdateGraphicControl()`. Doing so ensures that DAE's token system will operate properly.

• Implement `Idle()`. This is called by the process, telling the UI to take care of its idle processes.

• Implement `SetControlHighlight()`. This is called by the process, telling the UI to handle a change to the highlighting of a control.

• Implement `ChooseControl()`. This is called by the DAE to find the index of the control containing the point (in local coordinates) passed in.

Discussion and Known Issues

■ **Plug-Ins and Modifier Keys** There are some behaviors expected by users in regard to clicking on plug-in controls with certain modifier keys. The following behaviors are expected:

Mac Keys	Windows Keys	Expected Behavior
Command+Click	Control+Click	Adjust control's value with fine control
Option+Click	Alt+Click	Return control's value to default
Shift+Click	Shift+Click	Move stereo control values in tandem
Command+Option+Control+Click	Control+Alt+Start+Click	Activate popup menu for automation

Try any of these on any DigiRack plug-in and to see what these behaviors are supposed to do. In order to give the user the same experience with all plug-ins, it is important that you implement these behaviors in your plug-in. The Template_NoUI demonstrates the behaviors in yellow. The fine control behavior is accomplished by overriding the `VSTGUI::CSlider::mouse()` method. The return to default behavior is demonstrated by the `CNoUIControlListener::controlModifierClicked()` method in CVSTEditor.cpp. The automation popup menu behavior is possible on the Mac by overriding `CProcess::ChooseControl()` in CTemplateNoUIProcess. For all of these, please see the source code for further details.

Changes as of SDK versions after 7.0: Previously, the automation popup menu behavior had not worked in the Windows version of the Template NoUI plug-in. The reason for this is because the basic principles of the window systems on Mac and Windows are different. On Mac, when a mouse event occurs, it is first sent to the "super view" – the parent window of the window in which the click occurs. Normally, this event is then sent to each successive smaller view, until the event is handled. In this case, if Pro Tools detects that the three modifier keys are pressed, it handles the event itself, calling into the plug-in's `ChooseControl()` method to find out the index of the control. In contrast, the normal behavior on Windows is for an event to be sent directly to the child window, and if it is not handled there, it is sent to the parent window. Since this plug-in gets around the Mac2Win library's emulation of the Macintosh window system, the event is sent directly to the plug-in window, before Pro Tools can intercept it.

To get around this, you need to manually pass up the appropriate events to the Pro Tools event handler for Windows. This functionality has been added to the Template NoUI example plug-in. Specifically look at the WindowProc() function in vstgui.cpp. Take notice of the special event handling for the `WM_LBUTTONDOWN`, `WM_MEASUREITEM`, and `WM_DRAWITEM` messages. Also take notice of the caveats outlined in the comments around these messages. **These caveats are very important!** These caveats state that you should never pass a WM_DRAWITEM or WM_MEASUREITEM message up to the Pro Tools mesage loop unless you are sure that the menu was originated from Pro Tools. If you implement custom menus for your NoUI style plug-in, you will need to be able to tell if the WM_DRAWITEM message is received for your custom menu or one that originated from PT. One way to do that is to add some custom userData to the window or lParam argument that would identify the message as coming from your code. Passing up a message to PT that is not expected by PT could cause a crash or other unexpected behavior. Please see the code for the full explanation.

■ **Resources** Even though the Windows version of the Template_NoUI plug-in only uses resources compiled in its own resources, the .dpm.rsr file is still required. When a plug-in is loaded, the Mac2Win library throws an error and the plug-in will not load if a valid .dpm.rsr file is not found next to the plug-in binary. Therefore, at a minimum, a plug-in using this method must use the empty ThisPlugin.rsr contained in the WinBuild folder of the project.

There are two cases where a plug-in using this method must still compile Mac-based resources, swap them to the data fork, and bring them over to the PC as a normal plug-in does. These cases are:

1) The plug-in is a TDM plug-in using HCP copy protection. The DSP resources in this case can only be in the Mac-based resource.
2) The plug-in has legacy page tables (compiled as .r file on Mac) in order to work in versions of Pro Tools prior to 6.4.

The Template_NoUI plug-in does not fall into either category.

Gestalt for Template_NoUI based plug-ins

In Plug-In SDK 7.3, we have added a new gestalt to notify Pro Tools that a plug-in does not use the plug-in SDK View widgets for UI construction: `pluginGestalt_DoesNotUseDigiUI`. This gestalt will allow us to better handle event processing and and other features that must be implemented differently for plug-ins that do not use our provided GUI framework. Please define this gestalt if it is appropriate to your plug-in, since we will use it in the future to provide better compatibility for plug-ins that do not use our View widgets.

Appendices

Appendix A:
The Plug-In Library/
Effect Layer Reference Guide

Within this Reference Guide, methods labeled with **CALL** imply that they are just that -- they are calls that can be made by you to acquire information or define/initialize the state of the object. **IMPLEMENT** means that you can override or implement this virtual method to fulfill the requirements of the method or provide your own functionality. **OVERRIDE** is likewise, but is meant to remind you that you should first call the inherited method within your implementation.

All `Define` calls be made in the constructor of its intended class, unless otherwise noted.

CEffectGroup Level

■ Constructor Calls

CALL `void DefineManufacturerNamesAndID(const char *name, const OSType)`
Sets your manufacturer names and your unique 4-character manufacturer ID for the plug-in. Use the '\n' separator between the various names. The names must be listed longest to shortest in length. The maximum length for any name is 32 characters (including null terminator).

```
DefineManufacturerNamesAndID("Spork, Inc.\nSpork Inc\nSpork\n", 'Sprk');
```

CALL `void DefinePlugInNamesAndVersion(const char *name, UInt32 versionNum)`
Defines the names of the overall plug-in, and the version number.

```
DefinePlugInNamesAndVersion("Spork Tools II\nSporkTools2\nST2", 1);
```

CALL `void AddGestalt(long selector)`
Defines specific properties of the entire plug-in -- for all Effect Types.

- `pluginGestalt_SupportsDeckChange` Allows RTAS and TDM Types to be automatically swapped by DAE; see `CEffectType::DefineRelationID()`.
- `pluginGestalt_IsCacheable` Allows the plug-in to be cached for faster DAE loading.

■ Virtual Methods

IMPLEMENT `void CreateEffectTypes()`
This is where all the Effect Types for the plug-in should be created, defined, and then registered with the group using the `AddEffectType()` call.

OVERRIDE `void Initialize()`

This is a good place to register any Custom Views that will be used in the GUI(s) , or create any global data for the overall plug-in.

OVERRIDE `ComponentResult GetRelatedProcessTypeOfType`

```
                                                (SFicPlugInSpec *plugIn,
                                                 long deckType,
                                                 long numInputs,
                                                 long numOutput,
                                                 long inputStemFormat,
                                                 long outputStemFormat)
```

Get a related and compatible process type from the one we have. This is called when connections are to be added or removed, or deck type needs to be changed.

If you do not have a related plug-in that is able to read the chunk information from another, return `kCantFindPlugIn`. Otherwise, you should return `noErr`, and update the PlugInSpec with the match.

■ Non-MuSh TDM Methods

CALL `void DefineDspResourceAndMaxChannels`

```
                                (SInt16 dspResourceNum,
                                 SInt16 cardType, UInt32 coreType,
                                 SInt16 maxChannelsAt48k,
                                 SInt16 maxChannelsAt96k = 0,
                                 SInt16 maxChannelsAt192k = 0)
```

For a Non-MuSh plug-in, the DSP code resources that are present need to be defined via this call. In addition, the card type, core type, and the maximum number of audio channels/streams that the DSP can process, need to be associated with this code resource.

The `maxChannelsAt` represents the total number of channels (instances) that the DSP code, stored in resource number `dspResourceNum`, can run on the particular `cardType`. For an Onyx/Satchmo/MIX code resource, the `coreType` that the DSP code is intended to run on must be specified as well or left to the default.

Possible values of `cardType` are
`kMerleType` Runs on a DSPFarm card.
`kSatchmoType` Runs on a MIX Farm or MIX Core card.
`kGershwinType` Runs on an HD card.

Possible values of `coreType` are
`SATCHMO_DRAM_CORE_TYPE` Runs only on a DSP with DRAM.
`SATCHMO_SRAM_CORE_TYPE` Runs only on a DSP with External SRAM.
`ALL_CORE_TYPES` Runs on any DSP. (Use for Merle and Gerswhin.)

OVERRIDE `short GetTotalDSPCyclesAvailable(DSPPtr dsp)`
Override for plug-ins that need to track instantiation counts using a DSP cycle count metric. The default implementation returns absolute maximum cycle count at the current sample rate with no accounting for sample rate pullup or host command overhead. (See `CSurroundDownmixerGroup.cpp` for example implementation.) See also `CEffectTypeTDM::DefineDspCycleCounts(…)`.

CEffectType Level

■ Constructor

```
CEffectType Type(OSType typeID,
                 OSType productID,
                 UInt32 plugInCategory = ePlugInCategory_None)
```

where *Type* is either AS, RTAS, MuSh, or TDM. The typeID is a unique identifier for the particular EffectType. The productID is used to group together EffectTypes that can share save-and-restore settings. The plugInCategory parameter is a bit mask which helps describe its functionality and is used for hierarchical menus and in certain control surfaces.

```
CEffectTypeRTAS *eqCompressorRTAS = new CEffectTypeRTAS('tmAS', 'tmpl',
                                    ePlugInCategory_EQ|ePlugInCategory_Dynamics);
```

See the "Plug-In Categories" section of the **Plug-In Features** chapter for a detailed description of the categories or the FicPluginEnums.h file included in the Plug-In SDK.

■ Methods

CALL DefineTypeNames(const char *name)

Defines the list of possible names for the Type that can be returned when DAE request them. The plug-in developer can provide names of varying lengths using the '\n' separator. Names must be listed in order of longest to shortest in length.

```
myTypeRTAS->DefineTypeNames("Groovy Pitch\nGrvy Pitch\nGPitch\nGPth");
```

CALL DefineRelationID(OSType relationID)

The default RelationID for an EffectType is EffectLayerDef::GENERIC_RELATION_ID. This ID can be used to group together EffectTypes which are related and compatible to each other. More specifically, EffectTypes that share a RelationID should be the same basic plug-in, only differing in their stem formats, and/or platform (RTAS or TDM.)

This piece of information is used by Pro Tools when it needs to swap out a plug-in. For example, say we have the following scenario: A Mono-to-Mono EQ plug-in is inserted on a mono track. We then instantiate a Mono-to-Stereo Chorus on the second insert of that same track. We then go back to the EQ and change it to a Mono-to-Stereo plug-in. The Chorus plug-in, connected after the EQ plug-in has to change from Mono-to-Stereo to Stereo-to-Stereo, if possible. The Effect Layer will handle this by searching for the Stereo-to-Stereo EffectType that shares the same RelationID with the Mono-to-Mono EffectType. Likewise, if a session is switching from TDM to LE, Pro Tools will try to swap out a TDM plug-in with the matching RTAS version.

All EffectTypes which share the same RelationID must also share the same ProductID, since their save and restore information should also be compatible. In most cases, there will be a one-to-one correspondence between RelationIDs and ProductIDs. However, all EffectTypes of a particular ProductID might not necessarily share a common RelationID. For example, you might wish for a 1-band EQ to share settings with a 5-band EQ, where the 1-band's settings map back and forth to the first band on the 5-band EQ; however, you wouldn't want the mono 1-band EQ to swap out with a mono 5-band EQ. In this case, all your 1-band EQ's of various stem flavors and deck types would share a common RelationID, and the 5-band EQ's would have a different common RelationID.

```
myTypeMuSh71->DefineRelationID('rt71');
myTypeRTAS71->DefineRelationID('rt71');
myOtherCompletelyDifferentTypeMuShStereo->DefineRelationID('bizz');
```

CALL `AddGestalt(long selector)`
CALL `RemoveGestalt(long selector)`
These calls define, or remove, a property of your particular EffectType.

- `pluginGestalt_CanBypass` Enables the bypass button in the GUI window. If your plug-in does not support bypass, then it should not only not add this gestalt, but it should not support it in your `CMyProcessType::Gestalt()` method nor should it include `kCanBypass` flag in your `CMyProcessType::GetProcessTypeFlags()` method. In addition, because the default implementation assumes bypass support, you'll need to override the `CProcessType::GetIsBypassableByCategory()` function as follows:

```
ComponentResult CMyProcessType::GetIsBypassableByCategory(long /*aProcessIndex*/,
UInt32* outCategories)
{
    UInt32 bypassBits = 0;
    *outCategories = bypassBits;
    return (0);
}
```

- `pluginGestalt_SideChainInput` Enables the sidechain input option.

- **RTAS/TDM** `pluginGestalt_DoesntSupportMultiMono` Inhibits the plug-in from working in the multi-mono wrapper system of Pro Tools.

- **AS** `pluginGestalt_NeedsOutputDithered` DAE will apply default dither to output.

- **AS** `pluginGestalt_RequiresAnalysis` Indicates that the plug-in requires an analysis pass on the data before it can perform a processing pass. An example of a plug-in that requires an analysis pass is a normalize plug-in that needs to go through the data and determine the maximum sample value before it can perform a process pass.

- **AS** `pluginGestalt_AnalyzeOnTheFly` Indicates that the plug-in requires an analysis pass before processing, but additionally enables Preview for in this case. Must be used in conjunction with `pluginGestalt_RequiresAnalysis`. When the Preview button is pressed, the analyze pass is done first; then Process is called continuously until the Preview button is pressed again.

- **AS** `pluginGestalt_OptionalAnalysis` Enables the analysis button in the GUI window, which forces an analysis pass when clicked.

To clarify the 3 analysis gestalt, the following table shows the combination of Preview, Analyze, and Process buttons that will appear based on the gestalt used:

	Preview	Analyze	Process
RequiresAnalysis			X
AnalyzeOnTheFly	X		X
OptionalAnalysis	X	X	X

- **AS** `pluginGestalt_DisablePreview` Disables preview functionality.

- **AS** `pluginGestalt_DoesntIncrOutputSample` Indicates that the plug-in does not provide an output. An example of this would be a sample rate conversion plug-in that wrote out a file in a different format instead of using the audio format DAE would normally assign to a newly created file in the session. This is currently unsupported - data must always be output.

- **AS** `pluginGestalt_UsesRandomAccess` Plug-ins that use random access perform processes that do not have a linear, 1:1 relationship with the input and output buffers. For example, a reverse process uses random access because its input buffers do not align with its output buffers. Likewise, a time-compression/expansion plug-in also uses random access since it does not produce a 1:1

relationship between its I/O buffers. Similarly, a reverb or echo plug-in that produces a tail and has a longer output than input, are examples of random access plug-ins.

◆ **AS** `pluginGestalt_RequestsAllTrackData` If a plug-in uses random access for the input and requests all track data, it will be able to get all of the data that is on the track. Otherwise, only the selected track data is available; which in most cases is all that is required.

◆ **AS** `pluginGestalt_PlayListSource` Plug-in will read data from playlist files instead of from a connection buffer.

◆ **AS** `pluginGestalt_ContinuousOnly` Plug-in only processes on continous data, not region by region.

◆ **AS** `pluginGestalt_MultiInputModeOnly` For a plug-in that doesn't support stem formats, causes the Track Process Mode Selector button to go away. For a plug-in that supports a stem format, enforces that the number of tracks processed is equivalent to the stem format.

CALL `DefineSampleRateSupport(SInt32 sampleRateSelector)`
Declares the sample rates for which the Type is capable of operating. Possible selectors are shown.

```
enum {
        eSupportsNoSampleRates
        eSupports48kOnly,
        eSupports48kAnd96kOnly,
        eSupports48kAnd96kAnd192k
}
```

CALL TDM/MUSH `DefineSampleRateSupportForCardType(  SInt32 sampleRateSelector,`
`SInt32 cardType)`
Declares the sample rates for which the Type is capable of operating on a particular card type. Possible sample selectors are shown.

```
enum {
        eSupportsNoSampleRates
        eSupports48kOnly,
        eSupports48kAnd96kOnly,
        eSupports48kAnd96kAnd192k
}
```

The card type options are:
```
kMerleType
kSatchmoType
kGershwinType
```

CALL `DefineStemFormats( SInt32 inputStemFormat,`
`SInt32 outputStemFormat,`
`SInt32 sidechainStemFormat = ePlugIn_StemFormat_Mono)`
This call defines the stem format handled by the Type's input and outputs. Implicitly, the stem format also defines the number of input and outputs that the Type will use. Currently, the Pro Tools/DAE is limited to mono sidechain inputs. Defining a stem format for an AudioSuite plug-in...

```
enum EPlugIn_StemFormat {
        ePlugIn_StemFormat_Generic,
        ePlugIn_StemFormat_Mono,
        ePlugIn_StemFormat_Stereo,
        ePlugIn_StemFormat_LCR,
        ePlugIn_StemFormat_Quad,
        ePlugIn_StemFormat_LCRS,
        ePlugIn_StemFormat_5dot0,
        ePlugIn_StemFormat_5dot1,
        ePlugIn_StemFormat_6dot0,
```

```
        ePlugIn_StemFormat_6dot1,
        ePlugIn_StemFormat_7dot0,
        ePlugIn_StemFormat_7dot1
}
```

CALL `DefineGenericStemFormat(SInt32 numIO)`
For a generic algorithm that is independent of channel ordering, the generic stem format provides a way for the same plug-in to operate on differing stem formats that use the same number of channels. The generic stem format must be applied to both the input and output, and the number of inputs and outputs must be equal. Therefore, only one parameter is supplied for this call. It is the number of channels in and out of the Type.

CALL `AttachEffectProcessCreator(EffectProcessCreator theCreator)`
The Type must be able to locally instantiate EffectProcesses. Pass this call a function pointer to a static function that will create new Process objects.

```
CEffectProcess* NewTemplateProcessAS()
{
        return new CTemplateProcessAS;
}
```

CALL `PushBackMuShGenus(CGenusAdapter &genusAdapter)`
Defines the MuSh system properties for the Type. See the chapter **Writing TDM DSP Code** for more information on the MuSh system.

CALL `DefinePageTableName(const char *name)`
Each Effect Type should be associated with a particular set (generic, Procontrol, Control|24, etc.) of page tables, which are defined in the Resource Fork. If a page table name is not defined then the default behavior is to use its `typeID` (once converted to string). By defining your own name, you can link all similar Types, e.g. the mono version of a RTAS, AS, and TDM plug-in, to a single set of page tables. The string length of `name` is limited to 63 characters.

TDM-Only Methods

CALL
```
DefineDspCycleCounts( SInt32 numForegroundCycles,
                      SInt32 numBackgroundCycles,
                      SInt32 cardType = kSatchmoType,
                      SInt32 coreType = EffectLayerDef::ALL_CORE_TYPES)
```
By default a TDM Type has cycle counts of zero; therefore, the cycle count metric is inert when the Plug-In Library is determining whether a new process will fit on a DSP chip. `numForegroundCycles` equals the number of cycles required to process one sample of audio. Likewise, `numBackfroundCycles` equals the average number of "background" cycles required to process one sample of audio. Currently, the arguments `cardType` and `coreType` are ignored.

In pseudo-code, the equation that the Plug-In Library uses to determine whether the DSP has enough time for the Process is:
If `numForegroundCycles` + `numBackgroundCycles` + "CurrentTotalUsedCyclesOnDSP" is less than `CEffectGroupTDM::GetTotalDSPCyclesAvailable()`, then `dspHasEnoughTime = true`.

See also `CEffectGroupTDM::GetTotalDSPCyclesAvailable()`.

CALL `DefineChannelCount(SInt32 numChannels)`
The default definition of a TDM Type's channel count requirement is "`max(GetNumInputs(), GetNumOutputs())`." This call can be used to define an explicit channel count. In conjunction with

the information provided through `CEffectGroupTDM::DefineDspResourceAndMaxChannels()`, the Plug-In Library can track the "channel" counts on a DSP to determine whether it has enough room to fit another instance.

See also `CEffectGroupTDM::DefineDspResourceAndMaxChannels()`.

Note: In reality, both "cycle" counts and "channel" counts are arbitrary. But, they are meant to be abstractions of the resources available on the DSP (cycle time, DMA bandwidth, RAM, etc.) They are metrics used by the host side code, i.e, the Plug-In Library, to manage the instantiation of new Processes on new DSP chips. You are free to manipulate the personal meaning and units of these metrics however you see fit. They are wholely encapsulated within the plug-in and not referenced externally.

CEffectProcess Level

At the Process level, RTAS and AudioSuite audio streams are passed through "*connections*", which are enumerated starting with zero. In TDM, the nomenclature of "*ports*" is used; ports are enumerated starting with one.

■ General Methods

Initialization

IMPLEMENT `void EffectInit()`
SOURCE: `CEffectProcess`
The Process is unaware of certain of its properties at the time of its construction. Therefore, certain functionality is not possible within the Process constructor. Instead, defer these activities until `EffectInit()`. At this point the Process is now aware of its plug-in type (AS, RTAS, or MuSh), its number of inputs and outputs, and its sidechain input.

Getting State Information

The helper methods do not provide valid information within the constructor.

CALL `bool IsAS()`
CALL `bool IsRTAS()`
CALL `bool IsMuSh()`
These are used to differentiate between different Effect Types.

CALL `SInt32 GetNumInputs()`
CALL `SInt32 GetNumOutputs()`
Obviously, these calls return the number of inputs and outputs the EffectProcess handles, as defined via the Effect Type.

CALL `SInt32 GetInputStemFormat()`
CALL `SInt32 GetOutputStemFormat()`
These calls return input and output stem formats that were defined via the Effect Type.

Using Controls

CALL `AddControl(CPlugInControl *plugInControl)`
SOURCE: `CProcess`
Adds a `CPlugInControl` to the Control Manager. Refer to the chapter **The Graphical User Interface** for more information. Like `DefineDLOGId()`, `AddControls()`, and the following associated control methods, can be postponed until `EffectInit()` so that controls can be dynamically added based on the i/o configuration, etc. Keep in mind that the index number of a control is based on the order in which it is added!

CALL `DefineMasterBypassControlIndex(UInt32 index)`
SOURCE: `CEffectProcess`
If your EffectType defines the property `pluginGestalt_CanBypass`, then your Process must include a master bypass control which should be of type `CPluginControl_OnOff`. Note that the master bypass is automatically filtered out of save-and-restore functionality.

CALL `FilterControlIdOnRestore(OSType controlID)`
This method gives you greater flexibility when working with a new revision of a plug-in that may want to filter out certain controls from older Settings files (.tfx). *Not Implemented.*

CALL `ModifyControlIdOnRestore(OSType controlID)`
Not implemented.

CALL `FilterControlIdOnSave(OSType controlID)`
SOURCE: `CEffectProcess`
Allows specific controls to be filtered out of "Save Settings" functionality. This call is automatically invoked on the Master Bypass control when specified by the `DefineMasterBypassControlIndex()` call.

CALL `SetControlValue(long controlIndex, long value)`
SOURCE: `CProcess`
Dispatches a “token” to initiate the process of setting the specified control to a particular value. Refer to the **Graphical User Interface** chapter for more usage of `SetControlValue()`.

IMPLEMENT `UpdateControlValueInAlgorithm(long controlIndex)`
Anytime a control value is altered this method is invoked so that the algorithm can be updated with this new parameter information.

GUI/View

CALL `DefineDLOGId(SInt16 dlogId)`
A plug-in's window is defined within a DLOG item of the resource fork of the plug-in. This call specifies the ID of the DLOG item from which the dialog should be generated upon instantiation of the EffectProcess. (Optionally, this call can be delayed until `EffectInit()`, so it can be dynamically set based on the stem format, plug-in type, etc.)

OVERRIDE `void SetViewPort(GrafPtr aPort)`
Be sure to call the inherited `CEffectProcess::SetViewPort(aPort)` first. This allows the plug-in views to draw into the view port that DAE is passing in. After that, this method provides an opportunity

to attach any necessary views back into your Effect Process, e.g. meters. DAE passes in a NULL pointer for `aPort` if it is detaching the view port.

OVERRIDE `void DoTokenIdle(void)`
This is the `CProcess` method where all incoming tokens are processed to update the controls. Since it is called regularly and periodically, it makes a good place to update any special graphics that your Process might have, such as meters and "lights". Note: This call is only dispatched while the plug-in window is open, and therefore, cannot act as a general processing "thread" for the plug-in.

IMPLEMENT `ComponentResult DoSetCursor(EventRecord *theEvent)`
This method is called while the cursor is within the plug-in's window region. Here you are given the opportunity to modify the cursor. (See the `SetCursor()` MacOS API.) The return value of the method must be `kFicPlugInDidSetCursor` to inform DAE that the cursor was changed.

OVERRIDE `UpdateControlGraphic(long controlIndex, long aValue)`

User Input

IMPLEMENT `Boolean HandleKeystroke(EventRecord *theEvent)`
Overriding this function allows a plug-in to catch key events that are not automatically handled by the GUI. Keyboard shortcuts can then be implemented. Inform DAE that your plug-in has accepted and handled the key event by returning *true*; otherwise, return *false*.

Plug-ins should use the *control* key on Macintosh (*Start* key on Windows) as a modifier key. However, it is important to understand how Pro Tools handles key commands. Pro Tools first gives all open plug-in windows, starting with the window that has focus, the opportunity to handle an incoming key command. If no open windows handle the command, then Pro Tools will handle it itself.

There are two important implications of this system: 1) It is of vital importance that your plug-in return false as the default so that your plug-in does not “eat” important key commands intended for Pro Tools, and 2) If you do override a key command that is already used by Pro Tools, let your users know that the key command will therefore have a different effect depending on whether the plug-in window is open or closed. For example, if your plug-in implements *control+B* for Bypass, this same key command will split a region at the cursor if Pro Tools gets the command. A complete list of key commands used in Pro Tools is available in a pdf file called Keyboard Shortcuts, available in the release installations of Pro Tools. D-Show has reserved all *Start+key* commands for plug-ins.

The following keyboard shortcut sequences are automation shortcuts in Pro Tools, and therefore should not be used:

Command+Option+Control (Mac) / Control+Alt+Start (Windows)
Command+Control (Mac) / Control+Start (Windows)

Finally note, the following are standard keys for plug-in editing which are caught and handled by the Control Editors.

Command+Click (Mac) / Control+Click (Windows) = fine control value adjustment.
Option+Click (Mac) / Alt+Click (Windows) = return to default control value.
Shift+Click = move stereo control values in tandem.

Save/Restore Functionality

CALL `AddChunk(  OSType chunkID,`
`const char *description = NULL,`
`const SInt32 chunkDataSize =`
`EffectLayerDef::VARIABLE_CHUNK_SIZE)`

Use this method if your Effect needs to save-and-restore parameters. Subsequently, override the methods `SetChunk()` and `GetChunk()` to catch your `chunkID` and do the save and restore. Additionally, override `GetChunkSize()` if you do not specify a fixed and constant `chunkDataSize`.

OVERRIDE `ComponentResult GetChunk(OSType chunkID, SFicPlugInChunk *chunk)`

Override this method if you have added a custom chunk with the `AddChunk()` call . Here a chunk is being requested, so catch your `chunkID` and initialize the chunk's data which is pointed to by `chunk->fData`. `chunk->fSize` must be set to overall size of the chunk, including the header; this should be the same value that was returned by `GetChunkSize()`. Since settings files should be compatible between the Windows and Mac platforms, endian byte swapping is needed. Here, a imaginary byte-swapping routine `ByteSwapData()` handles this.

```
ComponentResult CMyProcess::GetChunk(OSType chunkID, SFicPlugInChunk *chunk)
{
     if (chunkID == MY_CHUNK_ID)
     {
          chunk->fSize = MY_CHUNKS_DATA_SIZE + sizeof(SFicPlugInChunkHeader);
          this->ByteSwapData(mSomeData);
          memcpy(chunk->fData, mSomeData, MY_CHUNKS_DATA_SIZE);
          return noErr;
     } else
          return CEffectProcess::GetChunk(chunkID, chunk);  // Not our chunk...
}
```

OVERRIDE `ComponentResult SetChunk(OSType chunkID, SFicPlugInChunk *chunk)`

Here, a save chunk is being passed into the plug-in. The plug-in should initialize to this new state. Like with `GetChunk`, if `chunkID` is owned by you, appropriately parse the chunks data, `chunk->fData`. Otherwise, pass the call onto the inherited class.

```
ComponentResult CMyProcess::SetChunk(OSType chunkID, SFicPlugInChunk *chunk)
{
     if (chunkID == MY_CHUNK_ID)
     {
          memcpy(mSomeData, chunk->fData, MY_CHUNKS_DATA_SIZE);
          this->ByteSwapData(mSomeData);
          return noErr;
     } else return CEffectProcess::SetChunk(chunkID, chunk); // Not our chunk
}
```

OVERRIDE `ComponentResult GetChunkSize(OSType chunkID, long *size)`

If `chunkID` is one of your plug-in's custom chunks, initialize `*size` to the entire size of your chunk in bytes. This includes all data bytes plus the size of the standard chunk header `SFicPlugInChunkHeader`. Otherwise, if you do not own this `chunkID`, pass the call to the inherited method. This method is invoked every time a chunk is saved; therefore, it is possible to have dynamically sized chunks.

```
ComponentResult CMyEffectProcess::GetChunkSize(OSType chunkID, long *size)
{
     if (chunkID == MY_CHUNK_ID) {
          *size = MY_CHUNKS_DATA_SIZE + sizeof(SFicPlugInChunkHeader);
          return noErr;
     } else
          return CEffectProcess::GetChunkSize(chunkID, size);
}
```

Processing

IMPLEMENT `ComponentResult GetDelaySamplesLong (long* numSamples)`

> NOTE: `GetDelaySamples()` has been deprecated in favor of `GetDelaySamplesLong()`. If you currently use `GetDelaySamples()`, please update your plug-in to implement the long version.

The total sample latency for a insert chain in Pro Tools is reported to a user when they Cmd-click (Ctrl-click on Windows) on the "vol" box within the mix window, changing the display to "dly". Pro Tools acquires this information from the plug-ins via this call. This information is additionally used for Pro Tools' Automatic Delay Compensation (ADC) feature.

Please see the "Automatic Delay Compensation" section of the **Plug-In Features** chapter for a detailed discussion.

OVERRIDE `ComponentResult SetSampleRate (long sampleRate)`
This method is invoked by DAE to set the current sample rate of the Process. With Pro Tools/DAE 5.3, the sample rate is fixed for a particular section, and this call will (should) only be made once during the initialization of the Process.

■ RTAS/AudioSuite Methods

IMPLEMENT `void ASInit()`
This method gets invoked after `EffectInit()`. Use it to initialize any AS or RTAS specific data.

IMPLEMENT `SInt32 ProcessAudio(bool isMasterBypassed)`
Return the number of samples that were processed. If `isMasterBypassed` is `true`, then move the audio unprocessed from the input to output buffer; otherwise, perform the processing algorithm.

CALL `DAEConnectionPtr GetInputConnection(SInt32 connectionIndex)`
CALL `DAEConnectionPtr GetOutputConnection(SInt32 connectionIndex)`
Given an index of the connection, these calls return the appropriate pointer to the DAEConnection structure. This structure is defined in `FicPlugInConnections.h`. Under RTAS, only the members `mBuffer` and `mNumSamplesInBuf` contain valid data.

```
// Plugin Manager Connection Parameter Block.
typedef struct DAEConnection
{
     long                mConnectionType;         // Type of connection being made (pluginType_TDM,pluginType_ASP...).
     DAEConnection       *mNext;                  // For internal use only.

     // The following fields are TDM specific.
     Ptr                      mTDMConnection;     // This is the old style of connection record.

     // The following fields are ASP specific.
     long                mBufferSize;        // Size of buffer in bytes.
     Ptr                 mBuffer;            // Location of buffer containing the data.
     long                mNumSamplesInBuf;   // Number of samples in buffer which have not yet been transferred.
     long                mStartBound;        // Start bound of all the samples passed through this connection.
     long                mEndBound;          // End bound of all the samples passed through this connection.

     SFicPlugInSpecPtr   mSourceSpec;        // Source of data going into the buffer.
     SFicPlugInSpecPtr   mDestSpec;          // Destination for the data.
     short               mOutputNum;         // Output number for the source data.
     short               mInputNum;          // Input number for the destination data.
     ConnectionFormat    mFormat;            // Format of the buffer data.
} DAEConnection, *DAEConnectionPtr;
```

OVERRIDE `void ConnectSidechain()`
OVERRIDE `void DisconnectSidechain()`
If any special functionality is needed as a result of the sidechain being connected or disconnected, implement it here.

CALL `SInt32 GetSideChainConnectionNum()`
If a sidechain connection is present, this call returns its connection number; otherwise returns `EffectLayerDef::NO_SIDECHAIN_CONNECTED`.

■ AudioSuite Only Methods

CALL `DefineDialogString(Int32 selector, const char *dialogText)`
This call allows a plug-in to better customize the text displayed by certain elements of the UI generated by the DAE application. Following is a list and description of the various elements, along with their associated selector values. Limit all string lengths to 255 characters plus a null termination (of course, you'll never need this much space!).

- `pluginStrings_Analysis` In the case of Pro Tools, this changes the "Analysis" button that appears in the plug-in window to the string you pass in. For example, a Normalize plug-in might want to rename this "Find Peak". A De-Hisser might change this to "Learn".

- `pluginStrings_MonoMode` In mono mode, we have one input per connection. In such an example, a Normalize plug-in might give back the string "Find Peak On Individual Track".

- `pluginStrings_MultiInputMode` In multi-input mode, we have more than one input per connection. In such an example, a Normalize plug-in might give back the string "Find Peak On All Selected Tracks". In a Stereo-to-Stereo process, it might pass back the string "Process As Stereo".

- `pluginStrings_RegionByRegionAnalysis` In Region-by-Region Analysis, we prime each region. In such an example, a Normalize plug-in might give back the string "Find Peak by Individual Regions".

- `pluginStrings_AllSelectedRegionsAnalysis` In an analysis pass across all selected regions, we prime as one large region. In such an example, a Normalize plug-in might give back the string "Find Peak Across All Regions".

- `pluginStrings_Preview` Rename or edit the Preview button.

- `pluginStrings_Progress` The plug-in receives a StringPtr containing the value that the DAE application wants to display in the progress dialog window during processing.

IMPLEMENT `void ModifyRegionName(char *regionName)`
When doing an AudioSuite "Process," DAE passes in the name it plans to give to a new region upon output. Here you are given the opportunity to insert your own name, or add a prefix or suffix. Limit your string length to 63 characters plus a null termination.

CALL `SInt32 GetASPreviewState()`
Can be used to determine whether AudioSuite is in preview mode or not. Returns `previewState_Off` if the plug-in is not in preview mode.

IMPLEMENT `UInt32 AnalyzeAudio(bool isMasterBypassed)`
Return the number of samples that were processed. If `isMasterBypassed` is `true`, then move the audio unprocessed from the input to output buffer; otherwise, perform the processing algorithm.

IMPLEMENT `ComponentResult InitOutputBounds(...)`

This method is invoked in several different cases: (1) before an analyze, process or preview of data begins. (2) at the end of every preview loop, or (3) after the user makes a new data selection in Pro Tools. The AudioSuite plug-in can then adjust the `mStartBound` and `mEndBounds` members of the output connection(s) to vary the length and/or start and end points of the outputted audio region.

IMPLEMENT `ComponentResult TranslateOutputSampleNum (short portNum, long outputSample, long *inputSample)`
Before every `ProcessAudio()` or `AnalyzeAudio()` call, the method `TranslateOutputSampleNum()` is invoked by DAE. Here, the Process is given the opportunity to specify its required block of input samples given an output sample number. (Warning: This method currently only works for portNum = 1. There is a bug for non-mono Processes. Bug#23590).

■ MuSh Methods

IMPLEMENT `void MuShInit()`
This method gets invoked after `EffectInit()`. Use it initialize your DSP parameters and any TDM specific data.

■ Non-MuSh TDM Methods

OVERRIDE `void SetDSPInfo(DSPPtr aDSPWeAreOn, CProcessDSP* aDSPObject)`
This method is invoked after the host command `hcStart` is issued to the DSP. This allows the Process to initialize the DSP algorithm prior to any TDM connections being made. If the DSP Process is being removed, NULL pointers are passed into this method.

CALL `SendControlValueToDSP(long portNum, long hostCommandNum, SInt32 value)`
CALL `GetValueFromDSP(long portNum, long hostCommandNum, SInt32 &value)`
Send/Retrieve a single 24-bit value to/from the DSP using the specified Host Command Interrupt number. Additionally, the port number is specified, which is translated into the underlying channel number before being sent to the DSP.

OVERRIDE `ComponentResult ConnectSidechainTDM(long cardChannel)`
OVERRIDE `ComponentResult DisconnectSidechainTDM()`
Implement the handling of the TDM sidechain here. `cardChannel` is the TDM card channel location of incoming sidechain audio stream. The Effect Layer does not have a default implementation for communicating this information to the DSP code; therefore, it must be implemented by the developer.

CALL `SInt32 GetChannelNumFromPortNum(long portNum)`
This call reports the channel number of the Channel Allocation System being used by the specified port number (1 = Left/Mono, 2 = Right, etc).

CProcessDSP Methods: Accessing the 56k DSP Host Port

The `CProcessDSP` object represents the host port of the 56k DSP. A pointer to this object is accessable through the protected member variable `fOurDSPObject` or the method `GetDSPPtr()`. Both are declared and defined in `CDSPProcess`.

CALL `bool Lock()`
This call as well as a corresponding call to UnLock() should be placed around any functions that access the DSP. Wrapping DSP access with Lock/Unlock will help ensure that the DSP is not being accessed by more that one code section at a time (read: Thread safe).

CALL bool Unlock()
See Lock()

Appendix B: Obsolete Items

CEffectGroupTDM Calls

`DefineDspResource` has been made obsolete by `DefineDspResourceAndMaxChannels`.

CALL `void DefineDspResourceAndMaxChannels`
`(SInt32 maxChannels,`
`short cardType,`
`short dspResourceNum,`
`UInt32 coreType = kTDM_IO_NoExtRAM_Core)`

For a Non-MuSh plug-in, the DSP code resources that are present need to be defined via this call. `maxChannels` represents the total number of channels (instances) that the DSP code, stored in resource number `dspResourceNum`, can run on the particular `cardType`. For an Onyx/Satchmo/MIX code resource, the `coreType` that the DSP code is intended to run on must be specified as well or left to the default.

Possible values of `cardType` are
`kMerleType` Runs on a DSPFarm card.
`kSatchmoType` Runs on a MIX Farm or MIX Core card.
`kGershwinType` Runs on an HD card.

Possible values of `coreType` are
`kTDM_NoIO_DRAM_Core` Runs only on a DSP with DRAM.
`kTDM_NoIO_SRAM_Core` Runs only on a DSP with External SRAM.
`kTDM_IO_NoExtRAM_Core` Runs on any DSP.

Type Gestalts

- `pluginGestalt_CanMute` Muting is no longer supported.
- `pluginGestalt_CanEdit` This gestalt is ignored.
- **RTAS/TDM** `pluginGestalt_SupportsActivate` This gestalt is ignored. All plug-ins must support the Activate/Deactivate system.

Appendix C: Deeper Into the Plug-In Library

CProcess Methods

Dealing with Controls

IMPLEMENT `ComponentResult GetNumControls(short *aNumControls)`

Inform DAE on the number of controls the plug-in has.

IMPLEMENT
```
ComponentResult GetControlValueInfo( long aControlIndex,
                                     long ControlValue,
                                     long selector,
                                     long* result)
```

Availability: Pro Tools 6.4 and later, except where noted; SDK 6.4 and later

`GetControlValueInfo()` allows the app to query a plug-in for the "meaning" of its control values. It was designed as a general purpose mechanism that will find additional uses with new selectors in the future. Right now, this API will be used:

- to ensure the EQ and Dynamics sections' EQ type selector LEDs light appropriately for a given band's filter type. We want the appropriate EQ type LED to light for each band's filter type – whether or not your band can switch between filter types.
- to ensure the state of the EQ section's In buttons matches the values of the associated plug-in controls. When each band's In button is lit, it must mean that the EQ is active/on/enabled. When it is unlit, it must mean that the EQ is off/disabled/bypassed. (Some plug-ins have EQ bypass controls (On = bypass); others have EQ In buttons (On = On). We can derive "meaning" from the control regardless of control value.)
- similarly, to ensure the state of the Dynamics section's Filt In buttons matches the values of the associated plug-in controls. When each band's Filt In button is lit, it must mean that the EQ is active/on/enabled. When it is unlit, it must mean that the EQ is off/disabled/bypassed.

`GetControlValue` is implemented at the Group, Type, and Process levels. See the Plug-In SDK sources for the implementation.

Selectors passed into `GetControlValueInfo()` will be of type `EControlValueInfo (FicPluginEnums.h)` and will allow for future queries on control value "meaning."

```
enum EControlValueInfo
{
// Filter bands in EDigi_EQPageTable, EDigi_CompLimPageTable, & EDigi_ExpGatePageTable
       eDigi_PageTable_EQ_Band_Type = 0,

// Values for filter IsInCircuit switches in EDigi_EQPageTable, EDigi_CompLimPageTable,
// & EDigi_ExpGatePageTable
       eDigi_PageTable_EQ_InCircuitPolarity,

// Called on a control in EDigi_EQPageTable to determine if we want to use the alternate
// control for that slot.  Currently this is implemented for the Q_Or_Slope controls, so
// that plugins can put a different slope control in the alternate slot and have it
// coexist with a Q control for each band.  This should be a runtime check based on what
// the band type is to determine which control should be shown.  If the Q and Slope
```

```
// controls are implemented as the same control object in the plug-in, this is not
needed.
        eDigi_PageTable_UseAlternateControl
};
```

Results (not return values) passed back by `GetControlValueInfo()` will be of one of these two types (in `FicPluginEnums.h`) for now.

```
enum EEQ_Band_Types                     // For eDigi_PageTable_EQ_Band_Type
{
        eIsHighPass = 0,                // Freq,       Slope
        eIsLowShelf,                    // Freq, Gain, Slope
        eIsParametric,                  // Freq, Gain, Q
        eIsHighShelf,                   // Freq, Gain, Slope
        eIsLowPass,                     // Freq,       Slope
        eIsNotch                        // Freq,       Q
};

enum EEQ_InCircuitPolarity              // For eDigi_PageTable_EQ_InCircuitPolarity
{
        eIsEnabled = 0,         // EQ band is in the signal path and enabled      (LED on)
        eIsBypassed,            // EQ band is in the signal path but bypassed/off (LED off)
        eIsDisabled             // EQ band is completely removed from signal path (LED off)
};


enum E_UseAlternateControl              // For eDigi_PageTable_UseAlternateControl
{
        eUseAlternateControl_No = 0,
        eUseAlternateControl_Yes
};
```

To add support for `GetControlValueInfo()`, you must to override `CProcess::GetControlValueInfo()`. For a given `EControlValueInfo` selector, control index, and control value, you must pass back a result (not a return value) denoting the "meaning" of that control value. If a control has no meaning in the context of the given selector, you should return `kControlValueInfoNotSupported`.

In the page table layouts defined in `FicHWController.h`, you'll notice several fields with comments that say `eDigi_PageTable_EQ_InCircuitPolarity` or `eDigi_PageTable_EQ_Band_Type`. You should pass back a valid result for at least each control mapped to these page table locations.

However, there are two notable exceptions to this rule. An EQ or Dynamics plug-in may have a band of EQ that does not include an EQ type selector control. The band is just always, say, an HPF. There's no control to map to the EQ type selector button in the page table and therefore no obvious control for which you would pass back `eIsHighPass` in `GetControlValueInfo()`. What is the solution? Well, the other controls on that band (Frequency, Q/Slope, Gain, In) know what type of band they're on. Thus the plug-in should pass back the relevant EEQ_Band_Types enum in `GetControlValueInfo()` for all controls on a given band of EQ. This also applies to key filter bands in your Dynamics plug-ins. In this way, the EQ type selector LEDs in both the EQ and Dynamics sections will always light appropriately.

The second exception applies to EQ or Dynamics plug-ins that have a band of EQ that does not include an In Circuit / Out of Circuit control for that band. In this case, the band is always In Circuit or On and the other controls for this band should return eIsEnabled as the result for `GetControlValueInfo()` when the `eDigi_PageTable_EQ_InCircuitPolarity` selector is passed in.

Note: The logic for the In Circuit / Out of Circuit LED is available in Pro Tools 6.7 and higher.

Code snippets for `GetControlValueInfo()` from the DigiRack EQ II 4-Band, EQ II 1-Band, and Dyn II Exp/Gate plug-ins are provided are provided here as examples:

```
/*============================================================================*/
ComponentResult CEQ4Process::GetControlValueInfo(long aControlIndex, long aControlValue,
                                                 long selector, long* result)
```

```
{
	ComponentResult err = kControlValueInfoNotSupported;

	if (selector == eDigi_PageTable_EQ_InCircuitPolarity)
	{
		switch (aControlIndex)
		{
			case kHSBypIndex:
			case kP1BypIndex:
			case kP2BypIndex:
			case kLSBypIndex:
			{
				if (aControlValue >= 0)
					*result = eIsEnabled;
				else
					*result = eIsBypassed;

				err = 0;
				break;
			}
		}
	} else if (selector == eDigi_PageTable_EQ_Band_Type)
	{
		switch (aControlIndex)
		{
			case kLSFreqIndex:
			case kLSGainIndex:
				*result = eIsLowShelf;		err = 0;	break;

			case kHSFreqIndex:
			case kHSGainIndex:
				*result = eIsHighShelf;		err = 0;	break;

			case kP2FreqIndex:
			case kP2BWIndex:
			case kP2GainIndex:
			case kP1FreqIndex:
			case kP1BWIndex:
			case kP1GainIndex:
				*result = eIsParametric;	err = 0;	break;
		}
	}

	return err;
}

/*============================================================================================*/
ComponentResult CEQProcess::GetControlValueInfo(long aControlIndex, long aControlValue,
                                        long selector, long* result)
{
	ComponentResult err = kControlValueInfoNotSupported;

	if ((selector == eDigi_PageTable_EQ_Band_Type) &&
		(aControlIndex == kFilterTypeIndex))
	{
		long	FilterType = IndexToFilterType(aControlValue);
		switch (FilterType)
		{
			case kFicLoShelf:	*result = eIsLowShelf;	err = 0;	break;
			case kFicHiShelf:	*result = eIsHighShelf;	err = 0;	break;
			case kFicParametric:	*result = eIsParametric;err = 0;	break;
			case kFicLoCut:		*result = eIsHighPass;	err = 0;	break;
			case kFicHiCut:		*result = eIsLowPass;	err = 0;	break;
		}
	}

	return err;
}

/*============================================================================================*/
ComponentResult CExpGateProcess::GetControlValueInfo(long aControlIndex,
```

```
                                    long aControlValue, long selector, long* result)
{
        ComponentResult err = kControlValueInfoNotSupported;

        if (selector == eDigi_PageTable_EQ_Band_Type)
        {
                switch (aControlIndex)
                {
                        case kHPFrequencyControl: *result = eIsHighPass; err = 0;   break;
                        case kLPFrequencyControl: *result = eIsLowPass;  err = 0;   break;
                }
        }

        return err;
}
```

The use of the alternate Q or Slope control requires some explanation. The DigiRack EQ III plug-in is used as an example. The EQ III plug-in has separate controls for Q and Slope in its HPF and LPF bands, depending on whether the band is set to notch or band pass. When the Band Type control is set to notch, the continuous Q control is used. When the Band Type is set to Hi/Lo Pass, the discrete Slope control is used. In the page table, the HPF / LPF Q controls are set to the "Q or Slope" in PeTE (`eDigi_EQPageTable_HPF_Q_Or_Slope`), and the HPF / LPF Slope controls are set to the "Q or Slope Alt" (`eDigi_EQPageTable_HPF_Q_Or_Slope_Alt`). Then, with the `GetControlValueInfo()` implementation below, the control surface properly swaps the controls when the band type control is changed. In other words, when the function is called with the `eDigi_PageTable_UseAlternateControl` selector, if the band type control (`mHPFRadioGroup`) is set to notch, then `eUseAlternateControl_No` is returned in the result and the control surface puts the Q control at that position. If the band type is set to pass, then `eUseAlternateControl_Yes` is returned and the Slope is placed at that position.

Note: The eDigi_PageTable_UseAlternateControl selector is available in Pro Tools 6.9 and higher.

```
ComponentResult C7BandProcess::GetControlValueInfo(long aControlIndex, long
                                        aControlValue, long selector, long* result)
{
        ComponentResult err = kControlValueInfoNotSupported;

        if (selector == eDigi_PageTable_EQ_Band_Type)
        {
                switch (aControlIndex)
                {
                case eControl_HPFRadioGroupIndex:
                        *result = (mHPFRadioGroup == eEQ_HPFNotchSwitch) ? eIsNotch :
                                                                        eIsHighPass;
                        err = 0;
                        break;

                case eControl LPFRadioGroupIndex:
                        *result = (mLPFRadioGroup == eEQ_LPFNotchSwitch) ? eIsNotch :
                                                                        eIsLowPass;
                        err = 0;
                        break;

                case eControl_LFRadioGroup:
                        *result = (mLFRadioGroup == eEQ_LFShelfSwitch) ? eIsLowShelf :
                                                                        eIsParametric;
                        err = 0;
                        break;

                case eControl HFRadioGroup:
                        *result = (mHFRadioGroup == eEQ_HFShelfSwitch) ? eIsHighShelf :
                                                                        eIsParametric;
                        err = 0;
                        break;

                }
        }
        else if (selector == eDigi_PageTable_UseAlternateControl)
```

```
	{
		switch (aControlIndex)
		{
		case eControl_HPFRadioGroupIndex:
		case eControl_HPFQIndex:
		case eControl_HPFSlopeIndex:
			*result = (mHPFRadioGroup == eEQ_HPFNotchSwitch) ?
					eUseAlternateControl_No : eUseAlternateControl_Yes;
			err = 0;
			break;
		case eControl_LPFRadioGroupIndex:
		case eControl_LPFQIndex:
		case eControl_LPFSlopeIndex:
			*result = (mLPFRadioGroup == eEQ_HPFNotchSwitch) ?
					eUseAlternateControl_No : eUseAlternateControl_Yes;
			err = 0;
			break;
		}
	}

	return err;
}
```

IMPLEMENT `ComponentResult GetControlNameOfLength( long aControlIndex, char* aName, long aNameLength, OSType inControllerType, FicBoolean *outReverseHighlight)`

`long aControlIndex` Particular control
`char *aName` The optimized name for the control (not to exceed aNameLength +1)
`long aNameLength` The requested length
`OSType inControllerType` The control surface the aName is destined for
`FicBoolean *outReverseHighlight` Flag to specify reverse highlighting for control surfaces that support it

> **NOTE**: If you are using the XML page table system (i.e. you edit your page tables with PeTE), the discussion below does not apply, since the short versions of your control names are stored in XML. You should still override this method but only return a value if the name length is 31. This will allow PeTE to get the long versions of the names from your plug-in which it uses to uniquely identify controls.

This method is used to get the 'name' of a control which has been optimized for the requested string length. These strings are used in various places including Plug-In view menus, Pro Tools automation graphs and control surfaces that support alphanumeric displays.

The returned string from `GetControlNameOfLength()` is placed in `aName`, which must not exceed `aNameLength` + 1 characters! Otherwise, it's possible to overflow the caller's buffer and crash the system! Therefore, any intermediate processing should be done in temporary buffers, and only at the end copy the temporary buffer back into `aName` being sure that `aNameLength` + 1 is not exceeded!

In addition to being a required override for a plug-in that has a View, automation and control surfaces need highly optimized string names according to the passed in `aNameLength`. As with 'value' strings in `GetValueString()`, certain expected lengths should be provided for in `GetControlNameOfLength()`. These lengths are: 3, 4, 5, 6, 7, 8 and 31. Please see the chapter on Hardware Controllers in the Plug-In Manual for more detailed information.

Note, when working with stereo plug-ins, the identifier 'Right' or 'Left' will often appear in the control name. For instance, 'Right Gain' or 'Left Delay.' When there is enough room to spell out the name completely, the identifier 'Right' or 'Left' should precede the parameter name (e.g., 'Right Gain' or 'Left Delay'). For short strings of the type that will appear on a control surface, it is preferable to place the

abbreviation of 'Right' or 'Left' ('R' or 'L') to the right of the parameter name. For example, if there are only 4 characters available to display 'Right Gain,' it should be shortened to: 'Gn R.'

Enumerations used here to reference control surfaces are defined in `'FicHWController.h'` in the Plug-In library. Please use these enumerations in your code when referring to a control surface, rather than 'hard coding' them. (E.g., when referring to the Mackie HUI, use 'cMackiePageTable' rather than 'MkTL'.)

IMPLEMENT `ComponentResult IsControlAutomatable(long aControlIndex, short *aItIsP)`

Set `*aItIsP` to `1` if automation should be enabled in Pro Tools for the control defined by `aControlIndex`. Otherwise, set `*aItIsP` to `0`.

IMPLEMENT `GetControlDefaultValue(long aControlIndex, long *aValuePtr)`

Return the default/initial value of the control.

CALL `ComponentResult SetControlValue(long aControlIndex, long aValue)`

The method is fully implemented in the Plug-In Library.

Appendix D: Glossary

AS See **AUDIOSUITE**.

ASIC Application Specific Integrated Circuit.

AUDIOSUITE A host-based, file-based plug-in architecture supported by DAE.

CARD CHANNEL A specific memory mapped address in a MIX DSP that is used to access a specific TDM time-slot.

CHANNEL ALLOCATION A semi-generic management scheme used by the Plug-In Library that allows multiple **DSP PROCESS** instantiations on a single DSP chip.

CONNECTION Connection is a word often used in this SDK. But, in the RTAS and AS realm it has the same implications as a **PORT** in TDM nomenclature. In contrast, connections are usually enumerated starting with zero.

DAE Digidesign Audio Engine.

DSI Sometimes refers to DigiSystemInit, the MacOS extension. However, "DSI" is often used to distinguish the newer shared library revision of DigiSystemInit. DSI was updated to DSI2 in Pro Tools 7.0.

DSPFARM A PCI-based TDM card utilizing Motorola 56002 DSP chips. This card is compatible with a MIX system via the TDM bus.

DSP MANAGER A shared library that plug-ins must interact with to manage DSP chip usage.

DSP PROCESS The conceptual instance of a plug-in running on a DSP chip.

EFFECT LAYER A collection of classes that inherit from the Plug-In Library, meant to streamline and simplify plug-in development.

GESTALT A physical, biological, psychological, or symbolic configuration or pattern of elements so unified as a whole that its properties cannot be derived from a simple summation of its parts.

GROUP The central class or object that manages all the **TYPES** that a plug-in might implement.

MERLE The production codename for the DSPFarm card.

MIX CORE A PCI-based TDM card utilizing six Motorola 56301 DSP chips.

MIX FARM Virtually identical to the **MIXCORE** card but provides addition DSP power. Multiple **MIXFARM** cards can be added to a MIX system.

MULTISHELL A DSP sharing technology that also greatly simplifies TDM DSP code development.

MUSH See **MULTISHELL**.

OSTYPE A misleading data type name that implies that it has something to the with the "operating system". It's simply a unsigned 32-bit integer.

PORT An audio stream connection to a TDM plug-in. Ports can be described in their physical sense, i.e. left port, right port, sidechain port and are enumerated starting with '1'.

PRESTO The codename of the DSP chip used on the HD Core and HD Process cards. It is of the 563xx architecture and runs at 100MHz.

PROCESS In a sense, a process is the plug-in as seen from the user's point-of-view. A process is instantiated for every plug-in inserted within the mix window, or selected from the AudioSuite menu.

PROCESS GROUP See **GROUP**.

PROCESS TYPE See **TYPE**.

REALTIME AUDIOSUITE See **RTAS**.

RTAS We pronounce it "r-tass," a.k.a. Real-Time AudioSuite. Processing for this type of plug-in is done entirely on the host CPU.

SATCHMO A.k.a. Louis Armstrong, or the codename of our MIX Core and MIX Farm DSP cards. See **MIX CORE**.

TDM Time-division multiplexing. A fancy acronym to describe how audio is pushed around our cards between DSPs.

TYPE A class/object which provides a description of the **PROCESS** it is capable of instantiating.

Appendix E: Document Revision History

V7.3

Chapter 4:

- Added “Manually Loading .tfx Settings Files” header to “Saving and Restoring Plug-In Settings” section
- Updated Drag and Drop section with new improvements added in Pro Tools 7.3
- Added “Feature: Pro Tools Session Browser Integration” section

Chapter 6:

- Updated Page Tables section to mention deprecation of legacy page tables.
- Added section on Control Mode types for rotary knobs.
- Added note regarding GetChunk() usage to “Saving & Restoring Custom Data” section
- Added new plug-in feature: Plug-In Streaming Manager to the Features section

Chapter 7:

- Added section on PCIe Accel Core and PCIe HD Accel Cards

Chapter 10:

- Added caveats regarding MIDI output nodes.

V 7.2

Chapter 4:

- Added section on Sidechain inputs.

Chapter 10:

- Added documentation for MIDI outputs (also see MicrobeSampler)

Chapter 17:

- Added notice of changes for NoUI to handle the Ctrl-alt-start-click event on XP.

V 7.0

Chapter 4:

- Added Drag-n-Drop Feature section
- Added a Saving Files of Any Type with a Pro Tools Session header to the Saving and Restoring Plug-In Settings section.

Chapter 8:

- Changed the Host Commands header in the Host to DSP Communication section to reflect function behavior changed in DSI2.

Chapter 10:

- Added DirectMidi Documentation.

Chapter 11:

- Changed CompareActiveChunk() section to reflect change in implementation

Chapter 12:

- Added Xcode porting info

Chapter 16:

- Added a memo on 321 DSP's in the "DMA Bug in the Presto DSP" section

Misc:

- Updated "all things MIDI" throughout the documentation

V 6.9

Chapter 3:

- Explained workaround for saving session specific custom chunk data in a chunk with the ID 'midi'.

Chapter 4:

- Corrected the rules for reporting RTAS delay in the Automatic Delay Compensation section.
- Changed the ADC bypass note: ADC is not recalculated when a plug-in is bypassed.
- Added 'Effect' category to the plug-in categories section.

Chapters 5 & 12:

- Updated Key Concepts (5) and ThisPlugin.rsr (12) sections with new resource swapping script information.

Chapter 6:

- Updated D-Control Emulator section.
- Updated Center Section Page Tables section: EQ type

Chapter 8:

- Added a section on the DSP Debugger and HCP copy protection.
- Added a warning about protecting the modifier register to the MultiShell II section.

Chapter 11:

- Updated "Plug-in IDs and Registering Your Plug-In With Digidesign" section.
- Updated DaeOpts.txt usage section

Chapter 15:

- Fixed some typos in Microbe chapter.

Appendix A:

- Updated HandleKeystroke() section.

Appendix D:

- Updated GetControlValueInfo() section

V 6.7.2

Chapter 12:

- Updated Data Endianess section.

Chapter 17:

- Updated the Template_NoUI section to include information for the new Mac version of the plug-in. The documentation here goes with the final 6.7 SDK version of the plug-in.

V 6.7.1

Chapter 1:

- Updated Hardware and Software requirements.

Chapter 2:

- Updated all of Chapter 2 with latest info.

Chapter 3:

- Updated all of Chapter 3 with latest info.

Chapter 6:

- Updated Alphanumeric Displays section.

Chapter 8:

- Updated DSP Debugger setup section.
- Removed DSP Probe Tool section.

NEW CHAPTER! Chapter 10: DirectMidi

- For now just a placeholder.

Chapter 11:

- Updated Plug-In icon section.

Chapter 13:

- Added the Sample Plug-In Technologies and Sample Plug-In Features charts.

Chapter 17:

- Updated chapter on Template_NoUI plug-in to reflect new version included in 6.7 SDK and above.

Appendix A:

- Updated information on HandleKeyStroke method.

Appendix D:

- Updated information on GetControlNameOfLength method.

V 6.7

Chapter 8:

- Added section on plug-in load ordering on a DSP.

Chapter 4:

- Added section on Auxiliary Output Stems.

Chapter 7:

- Added section on HD Accel hardware. This information mostly was copied directly from the GershwinIIDeveloper document, but contains a few changes.

Chapter 8:

- Updated the “Measuring and Verifying MuSh Cycle Counts” section with more complete information.

Chapter 11:

- Updated the “Debugging: Sending Debug Messages to the Console” section with more current information.

Appendix D:

- Updated the “CProcess Methods” section on GetControlValueInfo method.

V 6.4.3

Chapter 4:

- Updated the External Metering feature section to contain more detailed information on when meter and clip polling should take place.
- Added “Additional Considerations” section to External Metering feature section, explaining MeterTypes, MeterOrientations, and Metering Data Format expectations.

Chapter 6:

- Updated Digidesign Center Section Page Tables section.

V 6.4.2

Chapter 6:

- Updated Digidesign Center Section Page Tables section to make the standard layout of these page table types more clear.

V 6.4.1

Chapter 1:

- Updated all of Chapter 1!

Chapter 4:

- Added new chapter: Chapter 4: Plug-In Features, and bumped up all subsequent chapters.
- Added External Metering and Internal Clip section.
- Moved Automatic Delay Compensation section to this chapter and updated.
- Moved Plug-In Categories section to this chapter.
- Added Saving and Restoring Settings section.
- Added Plug-In Automation Section.

Chapter 6:

- Added reference to External Metering section in the Control Surface section.
- Moved Categories section from the Control Surfaces chapter to the Plug-In Features chapter.

Appendix A:

- Moved the detailed discussion of the Auto Delay Compensation feature to the Plug-In Features chapter.

Appendix E:

- Added Appendix E: Document Revision History.

V 6.4.0

Chapter 5:

- Updated Page Tables section to include information about new XML system.
- Updated Ethernet / Firewire / USB Controllers section to include D-Control, Command|8 and Center Section Page Tables sub-sections.
- Updated Implementing Page Tables section to include information about PeTE and XML system.
- Updated Color Highlighting Scheme sub-section to newest information.
- Updated Alphanumeric Displays section indicating usage of GetControlNameOfLength in new XML system.
- Updated Digidesign Extended Character Set section to indicate usage in PeTE.
- Updated Plug-in Categories sub-section of Other Advanced Control Surface Features section to include most recent categories and their descriptions.
- Added D-Control Emulator section.

Chapter 7:

- Updated MultiShell II section to include usage of MuShReady() method.
- Updated non-MuSh TDM Development section with ADC and GetDelaySamplesLong information.

Appendix A:

- Updated CEffectProcess Level section's description of GetDelaySamplesLong to indicate deprecation of GetDelaySamples and include information about ADC.

Appendix D:

- Updated CProcess Methods section to include information about new GetControlValueInfo() API.