Dear Digidesign Development Partners:

More than ever, Pro Tools offers "The Power to Create." You'll see that message wherever Digidesign announces and promotes the unique new music and MIDI features in our Pro Tools 7 product line. We're committed to provide you with timely and useful tools, info and services so that we can jointly deliver excellent stability, great user experience and upgraded functionality as Pro Tools itself advances in its features.

If you develop for the RTAS platform, whether you've been at it for a few months or a few years, whether you develop five Plug-Ins or forty, we would like you to actively go through this document. It offers a consolidated look at Digidesign's RTAS technology and provides background and ideas that will help you discover ways to fine-tune your products so that they can operate at their best on the Pro Tools platform. We're very pleased to be able to present this to you in one document, and we'll follow on with new information of this sort in the future.

We will match our "Power to Create" messaging with a specific commitment to prioritize your support inquiries as they pertain to the technical issues raised in this document. We're grateful for the work you do to develop fantastic RTAS offerings for the thousands of our customers who use your products with Pro Tools - and we are here to help. As always, you should send your technical questions to devservices@digidesign.com, or contact Jeff Matulich or myself to let us know how we can support you in your development, and in your business.

Yours very truly,

Ed Gray
Director, Partnering Programs

3/14/06

Fixing Performance Problems in RTAS Plug-Ins

Version 1.0 3/13/2006

In recent months, there has been an increased focus on RTAS plug-in performance here at Digidesign. We have strived to give our Pro Tools platform more host processing power, and to allow our users to run more RTAS plug-ins simultaneously. We recently made the RTAS engine multi-threaded, allowing RTAS processing to occur in parallel on multiple CPUs, and have continued to tune the engine for maximum efficiency at the smallest hardware buffer sizes.

Along the way, we have discovered a category of problems in some RTAS plug-ins that significantly hinder performance – especially at small buffer sizes. This document will explain the real-time nature of the RTAS engine, describe some common plug-in performance problems, and offer guidance and solutions that RTAS developers can put into use.

About the RTAS Engine

Pro Tools contains several important real-time host subsystems: disk I/O, MIDI, video, and RTAS, among others. Each of these subsystems is comprised of one or more high-priority threads that wake up at regular intervals to interact with hardware and/or other software components. Most subsystems have hard timing constraints. For example, the audio playback device will be forced to stop if the disk I/O subsystem has not finished reading a buffer of audio data by the time the audio is supposed to be played. The periodicity of each subsystem thread is also tuned to strike the correct balance between latency and efficiency. For example, making the audio device buffer too large results in unacceptable monitoring latency, while making it too small forces RTAS plug-ins to process a small number of samples at a time – increasing the average amount of processing time required.

The RTAS engine uses a separate high-priority thread for each available CPU upon which RTAS is enabled (the number of enabled RTAS CPUs can be adjusted in the Pro Tools “Playback Engine” dialog box). The RTAS engine has the highest priority of all the real-time subsystems, so all other Pro Tools threads have lower priorities.

The RTAS threads all wake up together at the beginning of each HW buffer period, and collectively work through the list of instantiated plug-ins – calling each one’s

ProcessData/RenderAudio function - to route the audio through the signal processing chain. In most cases, plug-ins are asked to process the same number of samples as the hardware buffer size, which generally results in maximum efficiency. Plug-ins that require 32-sample processing (i.e. that don't return true to the pluginGestalt_SupportsVariableQuanta gestalt) are called iteratively – 32 samples at a time – until a whole HW buffer's worth of audio has been processed. For best performance, it's important that plug-ins enable pluginGestalt_SupportsVariableQuanta – especially those plug-ins with significant per-call overhead. Processing the maximum number of samples in a single call amortizes this overhead and reduces the plug-in's average CPU consumption.

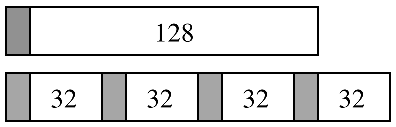

Figure 1. The top box represents a single processing period for a plug-in that processes 128 samples at a time. The shaded portion represents the plug-in's fixed per-call overhead (i.e. the amount of time the plug-in spends doing preparation before process actually begins). The bottom box represents the same plug-in being called four times (32 samples at a time) to process the same 128 samples. Note that the aggregate amount of time required to processing the same amount of data is significantly greater when pluginGestalt_SupportsVariableQuanta is not specified by the plug-in and the RTAS engine must therefore pass it 32 samples at a time.

It's important to note that an RTAS plug-in's ProcessData/RenderAudio function is called from within a high-priority RTAS processing thread. Most other calls into the plug-in are made in the low-priority main Pro Tools thread. Because calls into the plug-in can occur simultaneously in multiple threads, plug-ins must take care to synchronize access to internal data shared between code paths. Otherwise, difficult-to-debug problems can result.

Because the hardware buffer size is user-selectable between 64 samples and 1024 samples, the RTAS processing period ranges between 1.45ms and 23.21ms at the 44.1khz sample rate. The processing period represents a hard deadline: at 64 samples, the RTAS engine has an effective maximum of 1.45ms to call all the plug-ins and ensure that all processing for the period is complete. If the deadline arrives before the processing has finished, Pro Tools stops the transport and reports an error (the specific error code varies by device) to the user. The only alternative would be to allow the audio device to continue playing with a transient chunk of stale or non-existent audio.

Pro Tools allows the user to specify a maximum CPU usage limit between 45% and 99%, which creates a soft deadline for the RTAS engine. If the deadline is significantly exceeded (filtering ensures that a single transient spike does not constitute an excess), Pro Tools stops the transport and reports a -9128 error to the user. So at 64 samples with an 85% CPU usage limit, the RTAS engine nominally has 1.23ms to complete all processing.

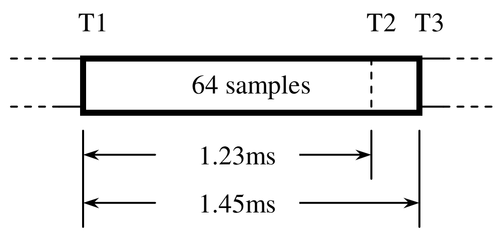

Figure 2. A single 64-sample processing period at 44.1khz with the user CPU limit set to 85%. The period begins at time T1 and ends at time T3 (which is also the beginning of the following period). The audio samples processed at T1 will need to be played at T3, so if the processing isn't complete by T3, an error occurs. T2 represents the user's CPU limit. If processing goes beyond T2 in multiple periods, a –9128 (CPU overload) error will result.

Performance Problems in Plug-Ins

It's clear that the timing constraints of the RTAS engine are fairly tight. But because the engine must call multiple plug-ins per processing period, the timing constraints placed on the plug-in's processing code are even tighter. If a single plug-in occasionally does something during processing that takes even an extra handful of microseconds, the result can be a user error. An occurrence that delays an RTAS processing thread by a millisecond or more is an extremely serious problem. One such occurrence is a threading phenomenon commonly referred to as a "priority inversion". There are many different things that can lead to priority inversion, and even a seemingly minor priority inversion problem in a plug-in's processing code can cause an RTAS thread to be blocked for an essentially unbounded amount of time.

Priority Inversion

Priority inversion occurs when a low-priority thread owns a shared resource that is needed by a high-priority thread. The high-priority thread must wait until the low-priority thread is finished with the resource, effectively "inverting" the priorities of the threads. Priority inversion would not be much of a problem if we could be assured that the low-priority thread would quickly relinquish the resource and allow the high-priority thread to continue. However, even code that is designed to achieve this cannot actually do so in practice. In open multi-threaded environments like Windows and OSX, there is inevitably some medium-priority thread that will preempt the low-priority thread while it still owns the shared resource. This causes both the high-priority and the low-priority thread to be blocked waiting for some number of unrelated medium-priority threads to finish executing – which takes an essentially unbounded amount of time. This scenario may sound farfetched or rare, but it can and verifiably does happen. In a real-time system like the RTAS engine, the worst-case performance – even if somewhat rare - is just as important as the average-case performance.

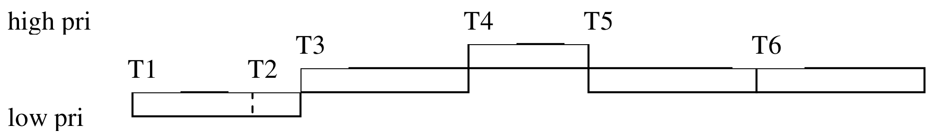

Figure 3. Priority inversion. A low-priority thread starts running at time T1 and acquires a shared resource at time T2. At time T3, a medium priority thread starts to run, preempting the low-priority thread still holding the shared resource. At time T4, a high-priority thread starts running, and at time T5 the thread attempts to acquire the shared resource. Because the shared resource is unavailable, the thread must block. The scheduler then runs the next-highest runnable thread, which is the medium priority thread. The medium priority thread may run for a long time, and it may be preempted by other medium-priority threads (as seen at time T6). This can continue for an unbounded amount of time, keeping both the low-priority and high-priority threads blocked.

Problem: Blocking Synchronization

Priority inversion problems in RTAS plug-ins commonly occur because some data object is shared between the processing code (ProcessData/RenderAudio) and the code that responds to control changes or UI interaction, or code running in a thread created by the plug-in. If blocking synchronization is used to coordinate access to shared data, priority inversion will result. By “blocking synchronization”, we mean any synchronization techniques utilizing OS-supplied objects that block a thread when contended for: locks, mutexes, semaphores, critical sections, conditional variables, spinlocks, etc. Simply stated, it is virtually impossible to call any blocking synchronization function from within ProcessData/RenderAudio without causing a potentially serious priority inversion problem.

```
void Function()
{
  EnterCriticalSection(&cs);
  ...
  LeaveCriticalSection(&cs);
}
```

Pseudocode Example 1. Even a simple-looking function like this can cause serious performance problems in an RTAS plug-in.

Waking up another thread from ProcessData/RenderAudio, if done carefully, will not lead to priority inversion. Typically this is done by incrementing a semaphore or signaling an event, and these operations do not normally cause the calling thread to block. However, for an important exception to this rule, please see the section below on the Apple OSX synchronization bug.

Solution: Non-Blocking Synchronization

To avoid priority inversion problems, there are several approaches that can be taken as an alternative to blocking synchronization. Non-blocking synchronization is usually built upon atomic operations provided by the CPU/OS, and allows multiple threads to share data coherently without the risk of priority inversion.

On Windows, the atomic APIs are in the InterlockedXXX family. On OSX, the BSD library provides a series of OSAtomicXXX functions. Of these, InterlockedCompareExchange and OSAtomicCompareAndSwap32Barrier are perhaps the most basic – allowing multiple threads to read/modify/write simple 32-bit variables without stepping on each other and without blocking. The functions compare a memory location with a supplied value, and exchange it with another value if there's a match. Typically, a thread will go into a loop, calling the function until the comparison is successful, indicating that the old value was atomically replaced with the new modified value. Repeated iterations of the loop are only necessary if another thread modified the variable between the time it was read and when the exchange was attempted.

```
long AtomicOr(long *location, long orMask)
{
  long orgValue, orValue;

  do
          {
          // get a snapshot of the original value
          orgValue = *location;

          // create the new value
          orValue = orgValue | orMask;
          }
  // try to replace the old value with the new
  while (InterlockedCompareExchange(
          location,orValue,orgValue) != orgValue);

  return orValue;
}
```

Pseudocode Example 2. This function atomically ORs new bits into a shared memory variable. Functions like this can be used to synchronize access to shared variables without the use of blocking synchronization.

If you need to synchronize access to a larger data object, one approach is to have multiple copies of the object, and atomically modify a 32-pointer to point to the "current" copy. If you need to stream data or messages between threads, a circular buffer can be created with atomically-updated read and write pointers. Alternately, a queue can be constructed from a linked-list in which the "next" pointers are updated atomically.

Inadvertent Priority Inversion

Priority inversion can have causes other than explicit block synchronization in ProcessData/RenderAudio. For example, some OS API functions internally use global state data protected by blocking synchronization. Memory allocators, for instance, usually have to coordinate the allocation requests of multiple threads, and do so via blocking synchronization. APIs which interact with hardware devices usually at some level have the capacity to block while waiting for access to the device. Ultimately, it's hard to know whether any particular API may block without being able to inspect the actual source code. Generally, OS designers may prioritize correctness over performance. Approaches that ensure correctness while allowing optimal performance –

like non-blocking synchronization – may not be widely understood and may be seen as hard to use. As a general rule, calling OS functions from within ProcessData/RenderAudio should be avoided. OS calls might instead be made from a low-priority thread that is woken up (carefully) from within the high-priority RTAS thread. It is important to carefully scrutinize any OS or library calls made in the RTAS callback in order to prevent accidental priority inversion.

The Apple OSX Synchronization Bug

As mentioned previously, waking up a lower-priority thread from within ProcessData/RenderAudio can occur without blocking the audio thread if certain care is taken. Although OS functions which increment a semaphore or signal an event do not typically block, a bug in the OSX POSIX Threads (pthreads) library makes some synchronization functions block unexpectedly. OSX contains a number of different threading and synchronization APIs, and all of them except the lowest-level Mach and BSD APIs are based on pthreads (this includes Carbon MP tasks and Cocoa NS threads, as well as pthreads itself) . The internal state of pthreads synchronization objects are themselves protected with user-mode spinlocks. This means that the simple act incrementing a pthread-based semaphore – an operation not conventionally expected to block – can contend for the object's internal state data and cause the thread to block on the spinlock. Obviously, this can be a serious source of priority inversion problems. The solution is to use a low-level Mach or BSD semaphore to wake up the thread. The BSD sem_open/sem_post/sem_wait functions are known to have good performance, and sem_post does not block the calling thread.

```
void RenderAudio()
{
  ...
  // let the background thread know
  // that there's work to do
  if (sem_post(mSemaphore) != 0)
          throw errno;
  ...
}

void BackgroundThread()
{
  while (1)
  {
          // Wait until there's work to do
          if (sem_wait(mSemaphore) != 0)
                  throw errno;

          // Do good stuff
          DoBackgroundStuff();
  }
}
```

Pseudocode Example 3. The BSD semaphore functions allow a thread to be awoken from the high-priority RTAS processing thread without the risk of priority inversion caused by the Apple OSX pthreads bug.

Summary

RTAS plug-ins, when loaded and instantiated in Pro Tools, run as part of a carefully tuned real-time system. As such, it is important to avoid the sorts of problems that can cause the real-time system to miss its deadlines and disrupt the user's workflows. Heeding the principles outlined in this document can help result in robust code which performs optimally even under the most aggressive time constraints.

References

Microsoft Windows atomic synchronization APIs:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/synchronization_functions.asp

Apple OSX atomic synchronization APIs:
http://developer.apple.com/documentation/Darwin/Reference/ManPages/man3/atomic.3.html

OSX threading architectures (fig. 3 shows the relationship of pthreads to other OSX threading APIs):
http://developer.apple.com/technotes/tn/tn2028.html