Using PAL Notifications

File: Using PAL Notifications.doc
Printed: 5/14/09 2:53:00 PM
Updated: 5/14/09 2:52:00 PM

Revision history

When	Who	What
2009-04-20	Brian Karr	Document Started
2009-05-01	Brian Karr	First Draft
2009-05-08	Brian Karr	PDF for release
2009-05-14	Brian Karr	Second draft, PDF for release

Table of Contents

1 Introduction

This document describes the Notification (Event/Listener) layer in the 3.4.2 and later versions of the Host sources. This layer operates between the Platform Abstraction Layer (PAL) and Host Applications such as command-line tools and GUI programs.

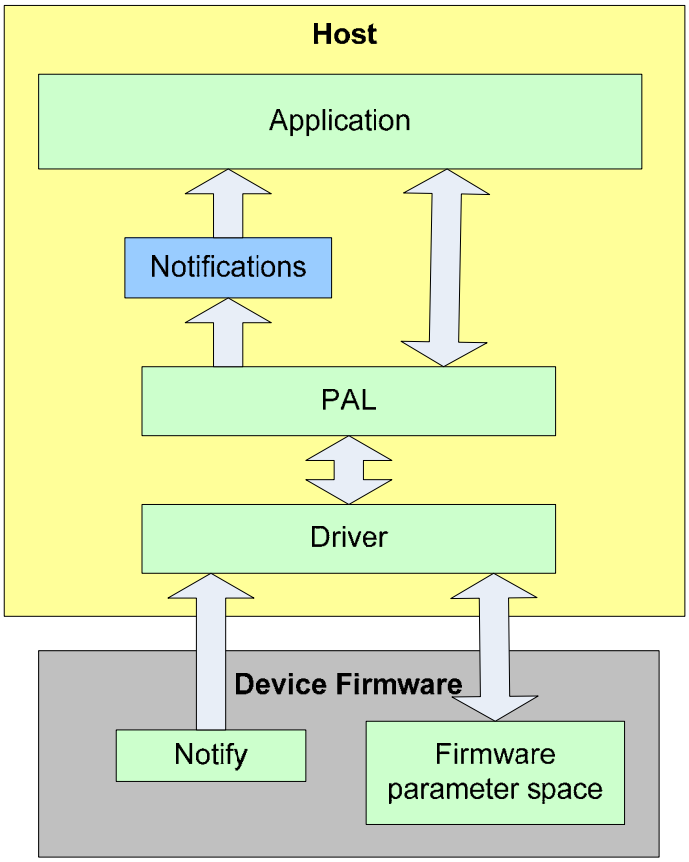
Previously, this functionality was incorporated within the Control Panel sources, and so could go unnoticed as a time-saving resource for developers. Since it is useful for many other applications, the source files for it have been logically separated into a separate directory in the source tree, the **`host/tools/pal_notify`** directory.

This document describes the use of the Notifications implementation.

2 PAL

The Platform Abstraction Layer abstracts platform differences into a common C++ API on all DiceDriver supported platforms. The PAL is documented in the source code itself with Doxygen tags, and the compiled html help from the source docs is available in the developer's subversion tag and online on the TCAT website.

The Notifications implementation is also a cross-platform C++ API, and adds the ability to more easily set up callbacks in application objects for notifications that originate in device firmware or the host driver.



2.1 The PAL parameter cache and callbacks

When the device firmware needs to notify the Host driver that something has changed, it sends a short message to the driver about which type of change happened, and the driver reads information from the corresponding parameter space in the device. The PAL then compares that with what it had previously cached, and then calls the relevant installed callbacks depending on the device parameter(s) that have changed.

Similarly for “bus” events, the developer can implement callbacks for various events common to all devices, or host related events. This includes settings such as sample rate, sync source, device arrival or removal, the changing numbers of running audio clients, etc.

The PAL provides classes with virtual callback methods for telling an object that a parameter has changed. The developer implements the virtual for each parameter of interest. In the callback, the object can then check the current value of the parameter. GUI programs usually have a thread that updates the UI, so usually the callback (which runs in a PAL thread) signals a waiting GUI thread that a parameter has changed. More on this below.

The PAL doesn’t include an implementation for handling the case where devices are added and removed dynamically on the 1394 bus. Most developers who use the PAL will implement some mechanism to implement parameter change callbacks as well as managing pointers or references for device arrival/removal, along with the usual resource cleanup that comes with it.

2.2 Reason for a notification “layer”

This implementation provides a simple mechanism for managing device and driver notification callbacks (notifications) in host applications, and automatically handles 1394 device arrival and removal and reference counting for automatic resource cleanup, and it’s straight-forward to integrate into multithreaded applications. In other words, it implements a lot of the underlying details most developers would need anyway.

The mechanism is not part of the PAL since the PAL is templatized and the notifications layer by design is not.

3 Event/Listener Notification Layer

3.1 Files

The notification layer is a directory containing events files and listener files, and optional EAP (Extended Application Protocol) device files. These files work with and require the PAL.

The event files implement the **`event_bus`** and **`event_device`** classes, which are used to contain and manage any attached listeners. The listener files implement the **`bus_listener`** and **`device_listener`** listener classes.

Bus listeners are used to handle “global” events such as sample rate, sync source, sync master, buffer size, channel mapping (Windows WDM), speaker map (Windows WDM) and operation mode changes.

Device listeners are used to handle device-specific events such as lock state, and nickname changes, etc. The device listener class, also provides functionality for uploading and downloading firmware.

The EAP files are used as an alternative device class when the developer wishes to support DICE-Jr and DICE-Mini based devices which have firmware that supports the Extended Application Protocol. This protocol adds additional functions for interacting with the hardware mixer and peak detector in those DICE variants.

3.2 General steps

- Include the **pal_notify** files, and the **PAL module** in your project.
- Derive any appropriate objects from **bus_listener** or **device_listener**. In each listener object, implement **mount()**, **unmount()** and any needed **update_xxx()** callbacks in each listener object.
- Create a singleton object that derives from the **system** class. Implement the **update_buslist()** notification callback.
- In the **update_buslist()** callback, determine if a valid bus exists, and if so **attach()** any bus listeners. On Windows a valid bus means a DiceDriver is installed, and there is at least one DICE device plugged into a 1394 bus. On MacOS there is always a valid bus as long as there is a driver installed and an active 1394 bus controller in the computer, even if there are no DICE devices on the bus.
- Once attached, the bus listeners **mount()** callbacks will be called. In the mount callback, call the parent class' `mount()` to initialize **m_bus** which gives you access to the event_bus accessor functions in the bus listener objects, for example `m_bus->clock()` is useful to use when the `update_clock()` notification arrives.
- The mount callback can also be used to attach any other `bus_listener` or `device_listener` objects in the applications object hierarchy.
- Once an object has been attached, it will receive `unmount()` callbacks whenever the bus device that corresponds to the reference it was attached to leave the bus, and it will then receive a `mount()` callback if that device is later plugged back onto the bus. The unmount callback then nulls the `m_bus` (or `m_device`) member, and any time the member is used in the object it is first validated to avoid exceptions.
- Bus listeners will receive `update_devicelist()` callbacks when the number of bus devices changes.
- The mount and unmount notifications can also be used in GUI applications to enable and disable corresponding GUI controls where necessary.
- **The listener object's `mount()` or `unmount()` functions are never explicitly called by the developer** (the *super* class' mount and unmount callbacks should not be confused here with the listener's mount and unmount. The super's mount and unmount should *always* be called by the listener.)
- Regarding threading, notification callbacks are called in a PAL thread, which uses synchronization locks. Callbacks will typically call PAL accessor functions in order to get any new values whose change is indicated by the callback. GUI programs will typically also use synchronization locks, so to avoid deadlocks the PAL callback will typically be used to signal a separate waiting GUI thread which then handles the new values that caused the event.
- When the object is destructed, the `detach()` function is called for you and all references are cleaned up. You will get an `unmount()` before this, which gives you a chance to do any other cleanup. If you wish to detach a listener object

which will stay instantiated, you can simply `detach()` it explicitly and it will stop receiving notifications.

4 Example

Developers can look at the Control Panel to see how the Notification layer is used. The examples below are from the TCAT Control Panel, which uses the JUCE[1] C++ class library. The code below is cross-platform.

4.1 system object

Below is an example system singleton class and some snippets that instantiate and use it.

4.1.1 Class implementation

Below is an example class which derives from the PAL system class `tcat::dice::system`, and in this case we specialize with the event_bus from the notification layer.

```
class TheSystem:
      public tcat::dice::system<event_bus>,
      public ActionBroadcaster
{
      typedef tcat::dice::system<event_bus> super;
      friend class ContentComp;

public:
      virtual     void  update_buslist()
      {
            super::update_buslist();

            // protect against changes to the bus list.
            tcat::dice::lock lock(this);

            // we use only one bus at a time, so you can decide how to
            // handle multiple busses. In this case we use the first
            // found bus.
            bool bFound = false;
            for (iterator bus=begin(); bus!=end(); bus++)
            {
                  if ((*bus)->size() > 0)
                  {
                        m_bus_id = (*bus)->id();
                        bFound = true;
                        sendActionMessage(T("bus changed"));
                        break;
                  }
            }
            if (!bFound)
            {
                  sendActionMessage(T("bus changed"));
            }
      }
```

[1] JUCE is available from Raw Material Software. For more info see http://www.rawmaterialsoftware.com/juce/index.php


```
private:
      tcat::dice::bus_impl::ID m_bus_id;
};
typedef tcat::dice::reference<TheSystem>  TheSystem_ref;
```

TheSystem object's job is to implement the update_buslist() callback. In the callback, you can iterate over the system deque of busses, and find one that has one or more bus devices. Multiple busses means there are multiple 1394 controllers installed in the computer. Here we simply pick the first bus that has more than one 1394 device connected, if any. We signal the GUI thread about this using a JUCE asynchronous mechanism. If no devices are found then a different action can be taken if the developer wants to handle that case separately, but here in either case we send the same action message. Note that this doesn't need an attach() or mount()/unmount() since it is always around.

Why only one bus at a time? The DiceDriver does not support audio streaming between devices on separate busses. This is due to the limitations if the PCI bus that 1394 controllers use, specifically it is not possible to provide low latency and sample-accurate synchronization at the same time. For this reason, it doesn't make sense to aggregate the two busses.

In your application, if you detect DICE devices on more than one bus, you can alert the User and let them know that they will only see devices on one of the busses, and to connect them all onto the same bus.

4.1.2 system class instantiation

The m_system member represents a singleton instantiation of TheSystem by virtue of the static creation method provided in the PAL. The example below shows the system class instantiation and subsequent checks for a valid driver, version matching between the driver and the PAL that is compiled with the application, and whether or not there is a bus with at least one DICE device on it.

```
try
{
      m_system = TheSystem::static_create<TheSystem>();
}
// make sure driver and PAL versions are compatible
catch(tcat::dice::xptn_driver_version xptn)
{
      tcat::exception::base * exception =
      dynamic_cast<tcat::exception::base *> (&xptn);
      String xptnStr = exception->usr_msg().c_str();
      AlertWindow::showMessageBox (AlertWindow::NoIcon,
            T(kTCAT_DICE_USR_STR),
            T("Error: "  + xptnStr
            + "\n\nThis application requires driver version "
            + TCAT_STRINGIFY(kTCAT_DICE_VERSION_MAJOR)
            + "." + TCAT_STRINGIFY(kTCAT_DICE_VERSION_MINOR)
            ".x\n\nPlease check the driver configuration."));
      JUCEApplication::quit();
}
// make sure the driver is installed
catch (tcat::exception::base exception)
{
      String xptnStr = exception.usr_msg().c_str();
      AlertWindow::showMessageBox (AlertWindow::NoIcon,
```



```
            T(kTCAT_DICE_USR_STR),
            T("Driver error: " + xptnStr + "\n\nCheck the driver
configuration or device connections."));
      JUCEApplication::quit();
}
```

Look for a bus with at least one device on it:

```
if (m_system)
{
      . . .

      // check if the bus list is empty
      if (0 == m_system->size())
      {
            AlertWindow::showMessageBox (AlertWindow::NoIcon,
                  T(kTCAT_DICE_USR_STR),
                  T("No devices found.\n\nConnect device(s)."));
      }
      else
      {
#ifdef __MACH__
            // MacOS always has a bus, so now further check if empty
            for (int i=0; i<(int)m_system->size(); i++)
            {
                  if (0 == m_system->at(i)->size())
                  {
                        AlertWindow::showMessageBox
                        (AlertWindow::NoIcon,
                              T(kTCAT_DICE_USR_STR),
                              T("No devices found.\n\nConnect
device(s)."));
                  }
            }
#endif      // __MACH__
            m_system->update_buslist();
      }
}
else
{
      // by now, if there is no system reference, then the driver
      // isn't loaded, so exit
      AlertWindow::showMessageBox (AlertWindow::NoIcon,
            T(kTCAT_DICE_USR_STR),
            T("Driver not found.\n\nPlease check the driver
configuration."));
      JUCEApplication::quit();
}
```

4.2 Handling a bus change

Below is the JUCE method of handling the action message sent above. This runs in the GUI thread. Most GUI frameworks will have a similar mechanism.

```
void actionListenerCallback(const String &message)
{
      if (0 == message.compare(T("bus changed")))
      {
```



```
            for (int i=0; i<(int)m_system->size(); i++)
            {
            // there is at least one bus (for WIN32 at least one
            //  device exists for which a driver is installed)
#ifdef WIN32
                  if (m_system->at(i)->size() > 0)
#endif

                  // assume all devices are on the first found bus
                  {
                        // attach the bus
                        event_bus_ref busref = m_system->at(i);
                        if (busref)
                        {
                              busref->attach(busWindow);
                              busref->attach(devWindow);
                        }
                        break;
                  }
            }
      }
}
```

The **busWindow** and **devWindow** objects are instances of classes that derive from bus_listener. Below are the BusWindow mount() and unmount() implementations.

```
void BusWindow::mount(tcat::dice::reference<event_bus> bus)
{
      // call super's mount() first to initialiaze m_bus
      bus_listener::mount(bus);

      if (m_bus)
      {
            // attach the bus listener views
            m_bus->attach(m_busgeneral);
            m_bus->attach(m_businfo);
      }
}

void BusWindow::unmount()
{
      // call super's unmount first
      bus_listener::unmount();
}
```

Here the mount() callback is attaching more objects that are managed by BusWindow. Those objects with then receive their own mount calls, and if the bus changes they will also see unmount calls. The unmount notification callback here simply nulls the m_bus reference.

Another important note is that the bus_listener doesn't need to do any cleanup related to Notifications. The destructor for BusWindow simply cleans up its JUCE GUI elements.

```
BusWindow::~BusWindow()
{
    deleteAllChildren();
```



```
}
```

4.3 Bus notifications

The m_busgeneral object above is a bus_listener that implements a number of callbacks relating to bus parameters. Below is the callback that handles changes in the bus sample rate.

```
void BusGeneralComponent::update_clock()
{
	if (m_bus)
	{
		m_rate = m_bus->clock();	// cache the current rate
		m_update |= updt_clock;
		triggerAsyncUpdate();		// notify the gui thread
	}
}
```

Here the callback asks the PAL what the current clock value is, and sets a bit in the m_update bitmap and uses another JUCE mechanism to asynchronously call another member function of the BusGeneralComponent class:

```
void BusGeneralComponent::handleAsyncUpdate()
{
	...
	if (m_update & updt_clock)
	{
		m_rate_cb->setSelectedId(m_rate+1, true);
		m_update &= ~updt_clock;
	}
	...
}
```

This allows the GUI thread to update its clock setting ComboBox to the current sample rate, avoiding any possible deadlocks.

4.4 A device listener

Below is an example of attaching a device listener. The LockStatusComponent class is a device_listener that displays the lock status of the sync master device. This is also an example of dynamically attaching and detaching a listener. The lock status component is always around, but will get notifications from the master device, which may change at any time.

The attachToMaster() function below is called by a bus listener when the master device has been changed. In this case, we detach it from the previous device and attach it to the new device. Note that detaching a device that hasn't yet been attached has no effect.

```
void LockStatusComponent::attachToMaster(event_bus_ref& busref)
{
	// no longer want events from the old device
```



```
	if (m_device)
	{
		m_device->detach(this);
	}

	// find the new bus clock master and attach
	event_device_ref devref = busref->find(busref->master());
	if (devref)
	{
		devref->attach(this);
	}

	// set initial value
	if (m_device)
	{
		// show the icon which represents the current lock state
		update_locked();
	}
}
```

As with bus listeners, the device listeners implement `mount()` and `unmount()` and they, at a minimum, will call the super class' `mount()` and `unmount()` to set and clear `m_device`.

4.5 Firmware Progress

The PAL device class provides a lot of functions for updating firmware flash files in DICE devices, and for reading the contents of a device to a file on the host computer.

As firmware is read from or written to a device, the developer would naturally want to give the user feedback for the progress of data transfer, and flash programming so they know not to pull the 1394 cable or remove power. The `firmware_progress()` callback can be used for that purpose. Firmware progress is a callback that occurs based on the percentage of completion of a task such as file transfer or programming data from RAM to flash.

Note that where possible these operations are atomic, which narrows the number of cases where an incomplete firmware image could be written to flash. For example, firmware is not written to flash until it has been completely loaded to the device's RAM and verified.

To see an example of how this is used, the `FirmwareProgress` along with the `DeviceSimpleFirmwareComponent` and Device `FirmwareComponent` implementations make full use of the firmware functionality.

4.6 EAP devices

For DICE devices that interact with firmware that implements the EAP protocol, the developer can include the **`dice_eap_device`** files from the pal_notify directory, and derive event_device from **`tcat::dice::eap_device`** instead of **`tcat::dice::device`**

This makes all of the EAP related notifications and accessor functions available for routing audio through the mixer, controlling the mixer parameters, and reading the peaks form the peak detector.

For an example of EAP metering, see the **`DeviceMeterComponent`**.

