Updating GUI Controls from your PlugIn RackAFX v6.8+
Will Pirkle

This document explains the RackAFX v6.8(+) API for updating GUI controls from your PlugIn. Previously this was done via a simple call to a plugin function called *sendUpdateGUI( )*. The problem is that this could potentially cause a race condition, depending on the timing of the GUI servicing. In v6.8, this paradigm was altered to provide a thread safe, and less confusing implementation.

Usage
Many users have requested the ability to update the GUI from within their plugins. Usually there are three reasons for this:

- having built-in presets within your plugin itself, separate from external presets from the other APIs
- allowing one control to alter the state of other controls, usually involving switches
- implementing the ability to share controls, for example a synth might have one set of envelope generator controls (Attack, Decay, Sustain, Release) and then a switch that allows those controls to connect to various destinations (amplifier, filter cutoff, etc…); when the user changes the switch, the knobs update to the proper positions

If you want to link continuous controls together (like knobs or sliders) so that they move at the same time, you should not use this updating mechanism as it is a clumsy way to accomplish this - instead, use the Advanced GUI API to create custom controls that your plugin owns and can manipulate*, or simply link them in the GUI Designer by giving them the same control tag.** If you really don't want to do any GUI programming but still need to link controls, see the example project ***LinkedControls for an example of linking two knobs controls together, however notice that you need to declare extra variables to keep track of button states for thread safety.

The ability to make outbound parameter updates exists in the other APIs as well, though the implementations vary; for AU you must generate AUEvents to the listener (the GUI), in VST3 and AAX you supply outbound parameter changes at the end of the audio process phase in a queue (VST3) or as parameter change calls (AAX). That being said, there are some who think that this kind of operation might suggest poor GUI design - for example, you could use a tabbed-panel of ADSR controls in the synthesizer example above, that would not take up much more space than the set of controls plus a switch, and would not require any of the following operations. So, use this functionality as you wish but be careful - **when presets are loaded and saved, any control that creates a GUI update will issue that update when the preset is loaded, so it is critical to make sure that your intended preset indeed requires such updating.**

GUI_PARAMETER Structure
RackAFX v6.8 introduces a structure that is used for thread safe operations between GUI and processing threads. This same structure is used to pass outbound parameter changes, either as meter values (built-in) or as GUI control changes that you implement from within the plugin. The structure is very simple and you only need to use two of its member variables, shown in **bold** below.

```
typedef struct
{
       int nControlIndex;         /* index of CUICtrl Object in plugin list */
       UINT uControlId;           /* RackAFX ControlId */
       float fNormalizedValue;    /* normalized version of parameter */
       float fActualValue;        /* actual value of parameter */
       bool bDirty;               /* flag that parameter needs update */
       bool bKorgVectorJoystickOrientation; /* flag */
```

```
	unsigned int uSampleOffset;/* future expansion */
}GUI_PARAMETER;
```

When UI controls change, your plugin will be queried as to any updates that need to occur. You will pass back a list of these structures, one for each GUI parameter that needs updating. **You will not alter your own underlying GUI-linked variables.** Instead, you will supply this outbound list of changes, very similar to the way it is done in VST3. NOTE: these calls to your plugin will be made from the low-priority GUI processing thread, not the high-priority audio processing thread. Your outbound changes will be handled in a thread safe manner and your your underlying variables will be updated during the pre-Processing phase, outlined in RAFX Technical Note 1, also in a thread safe manner.

The CLinkedList Object

There is a new generic C++ linked list object contained in pluginconstants.h which is used to create the lists of controls that you need to have updated as part of this operation. This object is tiny with short functions and low overhead. It is also simple for you to use although it is flexible enough to be used as a stack. The simplest way to show this is through an example. Consider this RackAFX GUI setup:

Control Name	Variable Name	RackAFX Control Id
Left Volume (dB)	m_fVolumeLeft_dB	8
Right Volume (dB)	m_fVolumeRight_dB	9
Unity Gain	m_uUnityGain	45
6db Boost	m_uBoost	46

The GUI consists of two volume knobs and two switches that are either SWITCH_ON or SWITCH_OFF in value (Unity Gain and Boost). The RackAFX Control Ids are shown and as normal, do not need to be zero-indexed. When updating the GUI, you will refer to these Control Id values, which are printed in a comment block above the two UI handler functions *userInterfaceChange( )* and the new *checkUdpateGUI( )* method. You will fill in the *checkUpdateGUI( )* method with any outbound changes that you want to occur as the result of other controls being changed. For this simple example, we will use the following logic:

Unity Gain

When this control is selected (in the SWITCH_ON position), we want the two volume knobs to move to their 0.0dB positions and the two underlying variables to change to 0.0 each.

6dB Boost

When this control is selected (in the SWITCH_ON position), we want the two volume knobs to move to their +6.0dB positions and the two underlying variables to change to +6.0 each.

The function prototype for *checkUdpateGUI( )* is:

```
checkUpdateGUI(int nControlIndex,
               float fValue,
               CLinkedList<GUI_PARAMETER>& guiParameters);
```

The parameters are:

```
nControlIndex:    the RackAFX Control Id of the control that has just changed

fVelue:           the actual value of the control in floating point format

CLinkedList<GUI_PARAMETER>& guiParameters:
                  a reference to the parameter list that you will populate
                  with outbound changes
```

In order to use the function, you follow these steps:

1. decode the incoming *nControlIndex* variable to check and see if the corresponding control will need to issue updates; <u>if the control does not create any updates, return false</u>

2. create a GUI_PARAMETER struct and fill in the information: the Control Id of the parameter to alter and the actual value it will take on after alteration, <u>then return true to indicate that updates are required</u>; NOTE: you do NOT need to alter the *bDirty* flag; create one GUI_PARAMETER structure for each control that needs updating

The CLinkedList has several member functions for placing items in the list and we only need to use one of them named *append( )* so for the example above, your function would look like this:

```
bool __stdcall YOURPROJECT::checkUpdateGUI(int nControlIndex,
                                           float fValue,
                                           CLinkedList<GUI_PARAMETER>& guiParameters)
{
      switch(nControlIndex) // decode the control that changed
      {
            case 45: // 45 is the Unity Gain button
            {
                  if(fValue == SWITCH_ON)
                  {
                        // --- change Left Volume to 0.0dB
                        GUI_PARAMETER param1 = {0};
                        param1.uControlId = 8;           // 8 = left volume knob
                        param1.fActualValue = 0.0;       // set to 0.0dB
                        guiParameters.append(param1);    // append it

                        // --- change Right Volume to 0.0dB
                        GUI_PARAMETER param2 = {0};
                        param2.uControlId = 9 ;          // 9 = right volume knob
                        param2.fActualValue = 0.0;       // set to 0.0dB
                        guiParameters.append(param2);    // append it

                        // —— return true since we made updates
                        return true;
                  }
                  break;
            }
            case 46: // 46 is the +6dB Boost button
            {
                  if(fValue == SWITCH_ON)
                  {
                        // --- change Left Volume to +6.0dB
                        GUI_PARAMETER param1 = {0};
                        param1.uControlId = 8;            // 8 = left volume knob
                        param1.fActualValue = 6.0;        // set to 6.0dB
                        guiParameters.append(param1);     // append it

                        // --- change Right Volume to +6.0dB
                        GUI_PARAMETER param2 = {0};
                        param2.uControlId = 9 ;           // 9 = right volume knob
                        param2.fActualValue = 6.0;        // set to 6.0dB
                        guiParameters.append(param2);     // append it

                        // —— return true since we made updates
                        return true;
                  }
                  break;
            }

            default:
                  break;
      }

      return false;
}
```

After the function is called, the host application will update the GUI locations and your parameter changes will be applied in a thread safe manner to be updated on the next audio processing cycle. At the end of the updating, your plugin base class will receive a call to the function *clearUpdateGUIParameters( )* in which it will clean out the linked list that you supplied. **There is nothing for you to do here as the clean up is automatic.** The reason that your plugin needs to clean out the list is that the *append( )* function calls use the *new* operator to create copies of the statically declared structures (remember this is happening in the low priority GUI thread, not the signal processing thread) and since they were created in your address space your plugin needs to delete them.

Notice the input control value *fValue* is a floating point type, and the outbound setting *fActualValue* is also a float. You can cast variables to and from your desired controls (int or UINT) as needed. In the above example, the outbound values happen to be floating point types.