# A1 Supporting Internal Presets

On occasion, you may want to control the GUI from within the plugin. Usually you want to pick up information from the GUI and use it to manipulate the GUI, by "invisibly" controlling knobs or buttons or linking controls so that they move together. There are a couple of ways to do this, one being the use of sub-controllers and custom-views. These are C++ objects you design with VSTGUI4 and can allow you to make amazingly cool controls and control clusters. If you are trying to intelligently link controls together, you should learn to use the VSTGUI4 sub-controller paradigm. It is the proper way to handle that problem, but you can also get away with it using the method.

If you want to implement internal presets, then this is a viable option that is relatively painless. In this app note, I will add internal presets to the *StereoDelayer* project and show you how to send a message to the GUI that will update all the controls at once. In addition I will use the same mechanism to turn on and off a LED that shows the dirty status of the GUI.

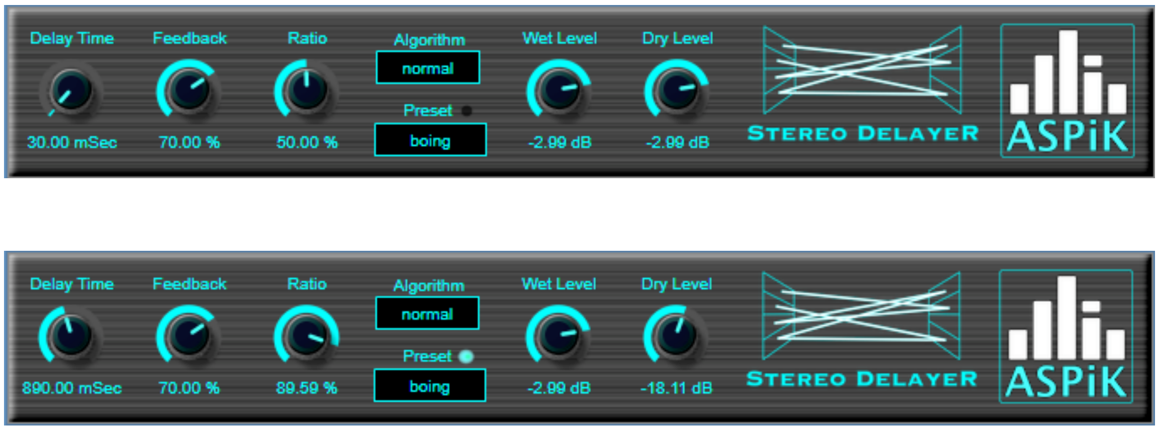


Figure A1.1: (top) a preset drop-list added to the ASPiK *StereoDelayer* project and (bottom) when the user moves any control, a "dirty GUI" LED lights up next to the "Preset" text, when the user selects a fresh preset, the LED turns off

## A1.1 Thread-Safe GUI Messaging

The ability to send thread-safe custom messages to the GUI has been part of ASPiK from the very beginning because it had been a request from users of the original (sunsetted v6) RackAFX software for a decade. The method here uses a thread-safe mechanism to deliver a message to the host plugin shell. That plugin shell then follows the thread-safe mechanisms built into the various APIs to issue messages that ultimately make the GUI controls move. This allows you to remotely control a GUI element (please, no hate mail from UX developers).

Note: your *PluginCore* can **NEVER** communicate directly with the *PluginGUI*, and your plugin core does not know, or need to know, if the GUI exists or not. The two objects do **NOT** hold pointers to each other.

The messages you are sending are to the plugin shell (the outer AAX, AU, VST or RAFX2 container object), which then updates the GUI according to its own thread-safe mechanism.

**Custom Views** may be used for pumping data into a GUI control, e.g. to show a waveform or FFT spectrum. These use a custom view interface and lock-free ring buffers that allow asynchronous, thread-safe communication directly with the GUI control, but they never communicate with the *PluginGUI* object itself. These are documented in the ASPiK SDK and demonstrated with several sample projects.

There are 2 things to remember:

- when you remotely set the position of another control, you send a message to the host that consists of a structure with information about which control you want to manipulate and the control's new value

- you let the host move the control, and then that information will but updated along with the normal GUI update cycle; in other words, just issue the GUI control command and don't change your underlying GUI-linked variables or parameters. These will change automatically on the next buffer process loop in the normal manner, as if the user had moved the controls

You only need to modify one function that you will find in your plugincore.cpp file called *guiParameterChanged( )* – you use this function to get notified when the user has manipulated a control, as well as sending a host command.

## A1.2 Internal Presets

The most common use of this mechanism is for supporting internal presets that are compiled into your plugin code. There are some things to consider.

1. When the user selects a preset from a list, you want the GUI controls to snap into the new locations. You have two options, one is to issue the GUI update command and let the normal buffer processing update cycle alter your parameters, as discussed above. If there is some crazy reason that the underlying parameters should also change at that instant – and I cannot think of one – then you can also alter the parameters as well using the same mechanism as the plugin shell uses when loading presets from a DAW.

2. This will circumvent the DAW preset loading system; although you see this in many commercial plugins, it may be scorned upon as dangerous. This is because

the preset is also a GUI control that will be serialized with the DAW session or DAW preset. **For some DAWs that still do not support VST3 Presets, this is the only option.**

3. If the user selects a preset and the controls snap to their new locations, then all is good. What happens if the user then moves one of the controls? You need a way to alert the user that the controls they are looking at are NOT the same as the preset that they are also looking at. Some DAWs place an * symbol next to the preset name, or other relatively benign way of letting you know the preset is "dirty."

Figure A1 shows how item #3 works – you can see a small aqua LED next to the "Preset" text that notifies the user when the GUI controls do not match the preset that is selected.

## A1.3 Internal Preset GUI Control

First, you should consider making the default GUI control values that you set when you created the controls (min/max/default) the very first present, named "default" or "factory init" or some other string that makes sense for the user. When the user opens the plugin for the very first time, the GUI will match the selection in the preset control. It will be vital that the GUI controls and the preset selection are synchronized when the user sees the GUI for the first time, and that they are also synchronized each time a preset is loaded.

You first need to add a GUI control for the user to select the preset. So, add a new string list parameter to your ASPiK project, or a new on/off switch to your RackAFX project. You can name the linked variable anything you like, and you need to remember the *controlID*, which is going to be based on your liked variable name. My control is setup as follows, notice that *default* is the first string in the list.

| | |
|---|---|
| Linked Variable: | **selectPreset** |
| String List: | **default, boing, gogo, whack** |

## A1.4 Preset Packaging

Next, you need to decide how to store the GUI control values for each preset:

1. Use the built-in *PresetInfo* structure and std::vector of these structures; see the *initPluginPresets* function

2. Use a custom data structure that goes with one of my C++ objects in the *fxobjects.h* file and described in my 2nd edition FX book; this is what I will use for this example

3. Use your own custom data structure that holds the information you need.

The *StereoDelayer* uses a single *AudioDelay* object for its entire operation. This object uses a custom data structure that holds all of the values of the controls on the GUI – it is perfect to use as a preset structure here. I have chosen 3 presets, plus the default preset for a total of four presets. The code is in the *plugincore.h* and *.cpp* files.

**plugincore.h**
At the top of the file, I have added some constants:

```
const uint32_t NUM_PRESETS = 4;
enum {DEFAULT, BOING, GOGO, WHACK}; // you need the default
```

In the class definition, I added the storage structures and a helper function to load the presets:

```
// --- storage for NUM_PRESETS worth of audio delay params
AudioDelayParameters presetParams[NUM_PRESETS];

// --- helper function to load the preset
void loadPreset(uint32_t index);
```

**plugincore.cpp**
I setup the presets in the *initialize* function in *plugincore.cpp*. This function will only be called once, whereas the *reset* function may be called whenever the sample rate changes. You may also use the constructor for this initialization. The reason I prefer the *initialize* function, is that it receives the path to the plugin DLL in the function argument and that path might be required for some custom stuff you need to load as part of a preset. This applies to synths that require folders of sample files, etc…

Notice how I setup the default preset first, using the current (default) parameter values, then the others are set manually with various preset values.

```
bool PluginCore::initialize(PluginInfo& pluginInfo)
{

      // --- setup presets
      //
      // --- DEFAULT
      presetParams[DEFAULT].algorithm = convertIntToEnum(
                                         delayType, delayAlgorithm);
      presetParams[DEFAULT].leftDelay_mSec = delayTime_mSec;
      presetParams[DEFAULT].feedback_Pct = delayFeedback_Pct;
      presetParams[DEFAULT].delayRatio_Pct = delayRatio_Pct;
      presetParams[DEFAULT].dryLevel_dB = dryLevel_dB;
      presetParams[DEFAULT].wetLevel_dB = wetLevel_dB;

      // --- short
      presetParams[BOING].algorithm = delayAlgorithm::kNormal;
      presetParams[BOING].leftDelay_mSec = 30.0;
      presetParams[BOING].feedback_Pct = 70.0;
      presetParams[BOING].delayRatio_Pct = 50.0;
      presetParams[BOING].dryLevel_dB = -3.0;
      presetParams[BOING].wetLevel_dB = -3.0;
      // --- long delay
      presetParams[GOGO].algorithm = delayAlgorithm::kPingPong;
      presetParams[GOGO].leftDelay_mSec = 1200.0;
      presetParams[GOGO].feedback_Pct = 60.0;
      presetParams[GOGO].delayRatio_Pct = 23.0;
      presetParams[GOGO].dryLevel_dB = -3.0;
      presetParams[GOGO].wetLevel_dB = -3.0;

      // --- massive FB
      presetParams[WHACK].algorithm = delayAlgorithm::kPingPong;
      presetParams[WHACK].leftDelay_mSec = 850.0;
      presetParams[WHACK].feedback_Pct = 97.0;
      presetParams[WHACK].delayRatio_Pct = 63.0;
      presetParams[WHACK].dryLevel_dB = -6.0;
      presetParams[WHACK].wetLevel_dB = -3.0;

      return true;
}
```

The *loadPreset* function does all the work and uses the *GUIParameter* structure that you need to understand first.

## A1.5 *GUIParameter* Structure

The structure for sending information back to the host is called *GUIParameter* and its contents are self-explanatory: the *controlID* is the ID of the control you want to remotely manipulate, and *actualValue* is the control's value, as a *double*, encoded as above. There is another option reserved for future use involving custom GUI data that you can safely ignore.

```
struct GUIParameter
{
      uint32_t controlID = 0;        ///< ID value
      double actualValue = 0.0;      ///< actual value

      bool useCustomData = false;    ///< custom data flag (reserved)

      // --- for custom drawing, or other custom data
      void* customData = nullptr; ///< custom data (reserved)
};
```

## A1.6 Sending the Host Message

You send updates back to the Host using an interface that is already built into your project called the *pluginHostConnector* – the conduit between the plugin and shell. Each GUI control you want to alter requires its own *GUIParameter* struct however, you may issue all of the control changes at once! The *GUIParameter* structures are sent to the host via a *std::vector* and you may send as many or few as you like. You may also send them in groups if that makes more sense for your application.

The *pluginHostConnector* will handle cleaning up the vector for you, so there is nothing else to do but issue the command. For example, to send a GUI control update message to a control with *controlID::myControl* and a new value of *123.45* you would setup the host message structure like this. This code declares the info structure, and sets the message to *sendGUIUpdate*, the only message currently supported.

```
HostMessageInfo hostMessageInfo;
hostMessageInfo.hostMessage = sendGUIUpdate;
```

Next, prepare a *GUIParameter* structure for the control with the new value you want to set:

```
GUIParameter param;
param.controlID = controlID::myControl;
param.actualValue = 123.45;
```

Then, add this structure to the vector and add as more controls as you like – it is OK to only have one control in the vector.

```
hostMessageInfo.guiUpdateData.guiParameters.push_back(param0);
```

Finally, use the interface to call the function – that is it, you are done.

```
// --- send message to host
pluginHostConnector->sendHostMessage(hostMessageInfo);
```

## A1.7 The *loadPreset* Function

This function will be called when the user alters the preset control. You pass the index of the preset and it sets up the *GUIParameter* structures according to the selection. The code is fairly self-explanatory here and only includes the preset selection; we will come back and add the code to turn the LED off, signifying a new preset load operation.

```
void PluginCore::loadPreset(uint32_t index)

// after validating the argument, you setup the structs

if (index >= NUM_PRESETS)
      return;

// --- grab the preset structure
AudioDelayParameters presetParam = presetParams[index];

// --- update the GUI
HostMessageInfo hostMessageInfo;
hostMessageInfo.hostMessage = sendGUIUpdate;

// --- setup a GUIParameter structure
GUIParameter guiParam[6];
```

```
// --- algorithm
guiParam[0].controlID = controlID::delayType;
guiParam[0].actualValue = enumToInt(presetParam.algorithm);
hostMessageInfo.guiUpdateData.guiParameters.push_back(guiParam[0]);

// --- delay time
guiParam[1].controlID = controlID::delayTime_mSec;
guiParam[1].actualValue = presetParam.leftDelay_mSec;
hostMessageInfo.guiUpdateData.guiParameters.push_back(guiParam[1]);

// --- delay FB
guiParam[2].controlID = controlID::delayFeedback_Pct;
guiParam[2].actualValue = presetParam.feedback_Pct;
hostMessageInfo.guiUpdateData.guiParameters.push_back(guiParam[2]);

// --- delay ratio
guiParam[3].controlID = controlID::delayRatio_Pct;
guiParam[3].actualValue = presetParam.delayRatio_Pct;
hostMessageInfo.guiUpdateData.guiParameters.push_back(guiParam[3]);

// --- dryLevel_dB
guiParam[4].controlID = controlID::dryLevel_dB;
guiParam[4].actualValue = presetParam.dryLevel_dB;
hostMessageInfo.guiUpdateData.guiParameters.push_back(guiParam[4]);

// --- delay wetLevel_dB
guiParam[5].controlID = controlID::wetLevel_dB;
guiParam[5].actualValue = presetParam.wetLevel_dB;
hostMessageInfo.guiUpdateData.guiParameters.push_back(guiParam[5]);

// --- send message to host
pluginHostConnector->sendHostMessage(hostMessageInfo);
```

That's it – this code will update the GUI controls, which will then be transferred into your plugin bound variables in the normal manner. If, for some reason you want to force the parameters to match, circumventing the system then you just call the internal function *setPIParamValue*.

```
setPIParamValue(controlID::delayTime_mSec,      delayTime_mSec);
setPIParamValue(controlID::delayFeedback_Pct,   delayFeedback_Pct);
```
etc…

## A1.8 *guiParameterChanged*

The plugin core object contains a function that is called anytime a user moves or touches a control. It is NOT designed for you to change anything regarding the internal state of the parameters – you can really screw things up badly if you misuse this function. The function is called each time any control is moved and you are given the *controlID* and the *value* of the control. Changing this *value* variable won't do anything – it is implemented with pass-by-value, but you may examine it and make decisions based on the control's new value. The first thing I do is check to see if the preset parameter has changed, and if so I call the *loadPreset* function.

```
bool PluginCore::guiParameterChanged(int32_t controlID,
                                        double actualValue)
{
      switch (controlID)
      {
            case controlID::selectPreset:
            {
                  // --- load a new preset
                  loadPreset(uint32_t(actualValue));

                  return true; // handled
            }
```

## A1.9 Status LED Code

That does everything you need and technically you would be finished, but we need to take care of the status LED as well. The status LED is implemented with a tiny custom control I coded into the *customcontrols.h* file. It is an on-off button with the mouse functions overridden and disabled. It is a one-way switch control that displays the LED using its *on* or *off* state. The reason I did not use a VU meter object has to do with how those objects get updated and animated during the *idle* function that is called every 50mSec to animate the meters, and also the controls when they are being automated. The meter will not respond exactly as we want, and in some DAWs you would need audio flowing just to see the updates. You can read more about creating and using custom controls in the ASPiK documentation so I won't bore you with it here. The status LED is added to the ASPiK core like any other control.

## A1.10 Custom Control: *CLEDControl*

You will find the code for this object in *customobjects.h* and it is ridiculously simple – an on-off button with the mouse disabled making it a write-only kind of control. The constructor just calls the base class and nothing else.

```
class CLEDControl : public COnOffButton
{
public:
      CLEDControl(const CRect& size,
                  IControlListener* listener = nullptr,
                  int32_t tag = -1, CBitmap* background = nullptr,
                  int32_t style = 0)
            : COnOffButton(size, listener, tag, background)
      {} // --- empty constructor, base class does all the work

      virtual CMouseEventResult onMouseDown(CPoint& where,
                                            const CButtonState&
                                            buttons) override
      {
            return kMouseEventHandled;
      }

      virtual CMouseEventResult onMouseUp(CPoint& where,
                                          const CButtonState& buttons)
                                          override
      {
            return kMouseEventHandled;
        }
};
```

I have also modified the *PluginGUI* object to include the new custom control (aka custom view). The code is inside of the *PluginGUI::createView* function. I gave this control a custom view string:

"CustomLED"

## A1.11 LED Resources and Plugin Parameter

We are going to treat this "trick" on/off switch as an ordinary ASPiK parameter. This will not only simplify coding, but will also allow the internal preset to be saved along with the DAW session data as well as its presets. Create a string list parameter is ASPiK, or an on/off switch in RackAFX. You only need the *controlID* of the parameter, which is the same as the C++ variable name, or *guiDirtyLED* for my project.

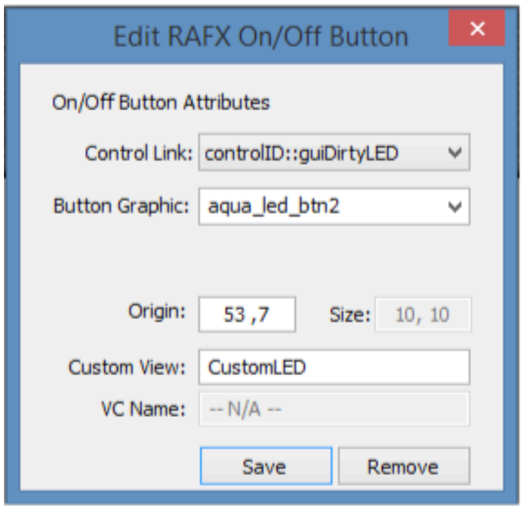
The status LED uses an on-off stitched image found in the *aqua_led_btn2.png* file. You need to add this to the ASPiK or RackAFX project in the normal fashion via the GUI Designer. So, add the resource and then add the on/off switch to the GUI, selecting the bitmap accordingly. Then, select the *controlID* that matches your ASPiK parameter.

***aqua_led_btn2***

Lastly, in the custom-view field or edit control, set the custom view name as:

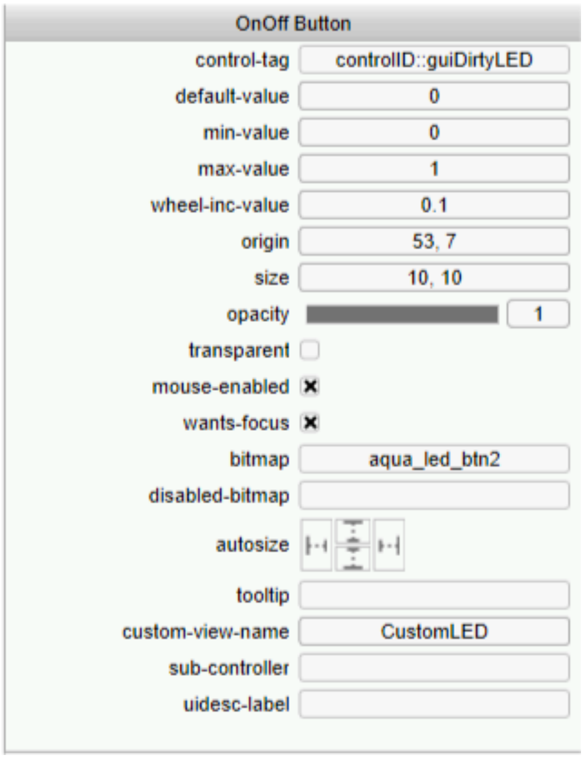
**CustomLED**

Figure A1.2 shows how this appears in the RackAFX and ASPiK GUI Designers. You need the *controlID*, the bitmap name, and the custom view name.


Edit RAFX On/Off Button

On/Off Button Attributes

| | |
|---|---|
| Control Link: | controlID::guiDirtyLED |
| Button Graphic: | aqua_led_btn2 |
| Origin: | 53 ,7 |
| Size: | 10, 10 |
| Custom View: | CustomLED |
| VC Name: | -- N/A -- |


OnOff Button

| | |
|---|---|
| control-tag | controlID::guiDirtyLED |
| default-value | 0 |
| min-value | 0 |
| max-value | 1 |
| wheel-inc-value | 0.1 |
| origin | 53, 7 |
| size | 10, 10 |
| opacity | 1 |
| transparent | ☐ |
| mouse-enabled | ☒ |
| wants-focus | ☒ |
| bitmap | aqua_led_btn2 |
| disabled-bitmap | |
| autosize | |
| tooltip | |
| custom-view-name | CustomLED |
| sub-controller | |
| uidesc-label | |

Figure A1.2: the on/off button attributes in RackAFX (left) and ASPiK (right) GUI Designers.

## A1.12 Status LED Code

You only need to alter two pieces of code to add the status LED implementation. First, when the user loads a fresh preset, the LED needs to turn off to indicate a proper preset. When any of the controls **except the preset control** is altered, the LED needs to turn on.

### LED Off State

For the first part, I add the LED off code to the vector of parameter structures during the parameter load operation in the *loadPreset* function, just before the host message call.

```
// --- delay wetLevel_dB
guiParam[5].controlID = controlID::wetLevel_dB;
guiParam[5].actualValue = presetParam.wetLevel_dB;
hostMessageInfo.guiUpdateData.guiParameters.push_back(guiParam[5]);

// --- turn OFF the dirty GUI LED
GUIParameter dirtyGUIParam;
dirtyGUIParam.controlID = controlID::guiDirtyLED;
dirtyGUIParam.actualValue = 0.0; // OFF
hostMessageInfo.guiUpdateData.guiParameters.push_back(dirtyGUIParam);

// --- send message to host
pluginHostConnector->sendHostMessage(hostMessageInfo);
```

**LED On State**

Lastly, I altered the *guiParameterChanged* function with a compound case statement for the rest of the GUI controls to turn on the LED if any control changes.

```
switch (controlID)
{
      case controlID::selectPreset:
      {
            // --- load it
            loadPreset(uint32_t(actualValue));

            return true; // handled
      }

      case controlID::delayFeedback_Pct:
      case controlID::delayRatio_Pct:
      case controlID::delayTime_mSec:
      case controlID::delayType:
      case controlID::dryLevel_dB:
      case controlID::wetLevel_dB:
      {
            // --- setup the message struct
            HostMessageInfo hostMessageInfo;
            hostMessageInfo.hostMessage = sendGUIUpdate;

            // --- setup a GUIParameter structure
            GUIParameter dirtyGUIParam;

            // --- turn ON the LED
            dirtyGUIParam.controlID = controlID::guiDirtyLED;
            dirtyGUIParam.actualValue = 1.0; // ON

            hostMessageInfo.guiUpdateData.
                              guiParameters.push_back(dirtyGUIParam);

            // --- send message to host
            pluginHostConnector->sendHostMessage(hostMessageInfo);

            return true;
      }
etc…
```

That's it – you now have code for the preset loading and status LED to get you started. Be sure to think about any other ramifications. All ASPiK parameters are automatable and if someone automates the plugin here, all will be OK because of the way I avoided altering the plugin parameters directly, and let the GUI control change do all of the work.

Will Pirkle

June 28, 2020 v1.0