Audio and GUI Thread Management in RackAFX, AU, VST3 and AAX

Will Pirkle

This document explains the RackAFX v6.8(+) API for processing audio and handing GUI updates in a thread safe manner. It also shows how the same process works for AU, VST3 and AAX. Please note that the latter are somewhat simplified to avoid implementation details.

RackAFX

In v6.8 and above, the paradigm for handling GUI control changes while processing audio changed to a thread safe method. The Focal Press book code does not change and the implementation details are done mainly at the base class level - as the plugin author, you will not even see the details unless you want to. The fundamental idea here is to make sure that the GUI and audio processing threads do not compete to access a plugin's internal variables to alter their states which could create race conditions. We also need to make sure that inconsistent-data conditions do not exist, in which the GUI and processing threads have different values for linked variables (by linked, I am referring to the fact that GUI controls are linked to underlying variables). Figure 1 shows how this works in RackAFX. The numbers here follow the numbers in the figure.

First, the RackAFX document object which contains the processing thread owns a fixed array of GUI_PARAMETER structures. There is exactly one structure for each GUI control. The parameter array is protected via a CriticalSection - a Windows OS structure that is used for thread synchronization. CriticalSections have very low processing overhead and are preferred over mutexes for this application. CriticalSections also have a "try and enter, return if you can't" function so that there is minimal impact on the audio processing.

Next, the original audio processing paradigm has been broken into three phases: pre-processing, audio processing, and post-processing. Comparisons with the other APIs will show that this three-part method is essentially baked into each of the different paradigms. The result is that in v6.8 and above, your plugin is responsible for setting its underlying GUI-linked variables and calling *userInterfaceChange()* in the pre-processing phase. In the post-processing phase, your plugin writes out any outbound parameter values, which are currently limited to meter values. This guarantees that your plugin's GUI control variables can not be changed while you are processing your audio data. Extra security measures have been included so that you can not corrupt any data during the pre and post-processing phases.

Updating the GUI from within your plugin is done in a new thread safe manner, described in RAFX Technical Note 2.

The code in the FX and Synth Books from Focal Press is still valid and does NOT change. This document describes details that are "behind the scenes" in the underlying CPlugIn base object and the RackAFX client. If you are following along in the books, you will not need to alter anything in your projects. This document is mainly for people interested in how RackAFX (and the other APIs) handle GUI parameter changes.

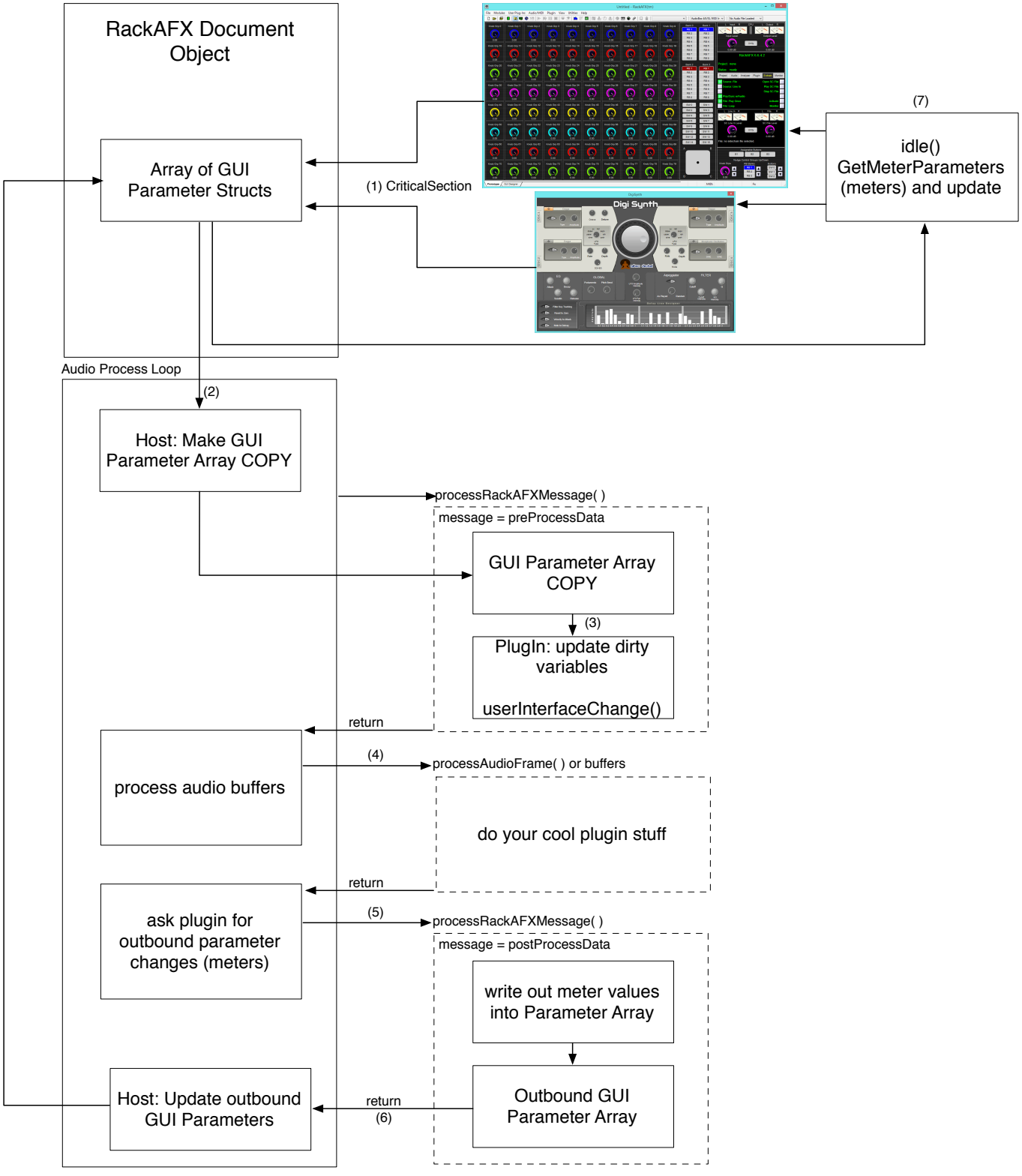

Figure 1: Audio/GUI processing in RackAFX

From Figure 1, we can start at the top and discuss what happens after a GUI change is made, then how the signal processing function works. When either the RackAFX main UI or your custom GUI makes a change via a user input, the new parameter value is written into the corresponding GUI_PARAMETER structure for that control and a “dirty” flag is set in the structure. A CriticalSection is used for thread synchronization for accessing this array. This is shown in (1) in the figure. The array of GUI_PARAMETER structures is created when your plugin is loaded into memory and is fixed in size - there is one GUI_PARAMETER structure for each RackAFX ControlId value - i.e. one structure for each of your plugin’s GUI-linked variables.

(2) At the start of each audio processing loop (the processing of one buffer of audio, in and out of the application), the document makes a fast memory-copy of the current array into a second array which is also fixed in size and created when the plugin is loaded.

A new function has been added to the RackAFX API which will greatly extend the functionality for future versions. This function is called *processRackAFXMessage()* and is implemented on the CPlugIn base class - you do not need to alter this function, though you may override it if you wish. The messages currently used are *preProcessData* and *postProcessData* though more may be added in the future.

(3) In the pre-processing phase, any parameters that are marked dirty in the list are transferred into your plugin variables, and *userInterfaceChange()* is called for each one. This is done in your plugin object, during the audio processing loop, but **before** any audio processing occurs. Notice that this is operating on a copy of the parameter array so the GUI is free to make other changes during the audio processing loop without interfering with your internal variables. This sequence guarantees that your plugin’s GUI-linked variables will not change during the audio processing operation and is common throughout the other APIs. It also guarantees that your plugin updates its own internal variables on the audio processing thread, rather than having them updated externally by the GUI thread.

(4) After the pre-process phase is complete, the usual audio processing function occurs, calling *processAudioFrame()* or either of the process-buffer functions depending on your personal choice. During this phase, you are free to access or even alter your own GUI-linked variables without worrying about the GUI thread accessing them as the GUI can only access the array of GUI_PARAMETER structures.

(5) After audio processing, the client will query you for any outbound parameter updates, specifically for metering variables. Again, this is all handled on the CPlugIn base class and you do not need to alter or deal with this code. The metering values are written into an array of GUI_PARAMETER structures that is passed by reference into the function.

(6) The updated meter variables are then applied to the document’s GUI_PARAMETER structures.

(7) Lastly, the meter values are updated during the GUI’s *idle()* processing loop, reading from the appropriate GUI_PARAMETER structure.

RackAFX v6.8 now implements an optional Automatic Parameter Smoothing operation to remove clicks, zipper noise, or other discontinuities from the GUI control variables as the user rapidly moves the controls around. This smoothing operation is also implemented in the derivative projects (Make VST, Make AU, Make AAX) as well as the RAFX-DLL-as-VST-DLL. This is detailed in RAFX Technical Note 3.

The following figures and descriptions show the similar processes that take place in AU, VST3 and AAX although there is some simplification here to avoid getting buried in the implementation details.

Audio Units

The AU API uses a map of parameter objects called Global Parameters to handle GUI control functionally. There is one Global Parameter object for each of your plugin's GUI controls (and linked variables). There is no thread synchronization object used as the authors did not want to incur the processing overhead of a mutex lock in MacOS. By this token, some might say that AU is not thread safe, however if you write your plugin according to AU guidelines, it will be thread safe by design. Atomic data accesses provide part of the data protection mechanism.

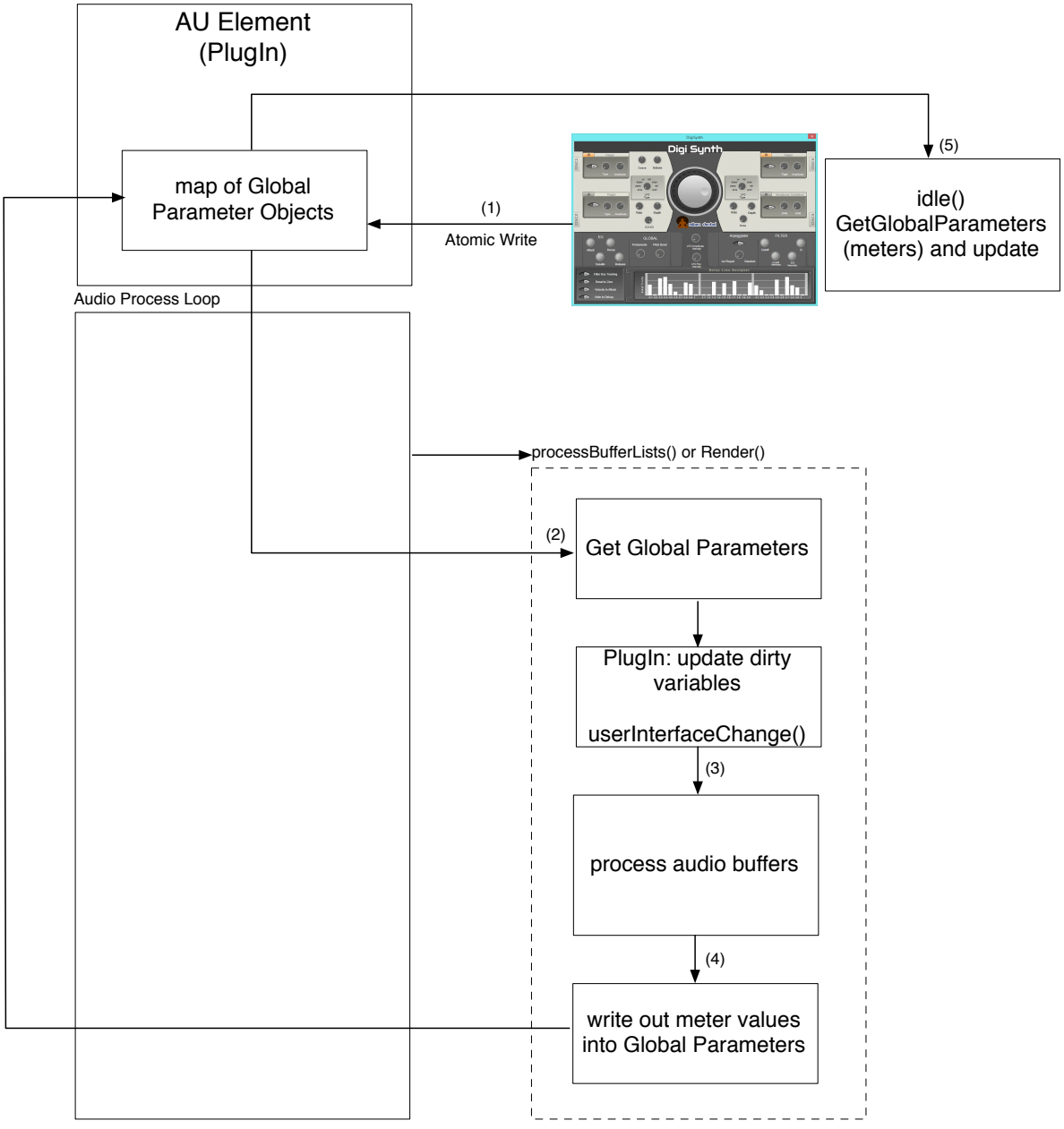

Figure 2: Audio/GUI processing in AU

(1) Starting at the top of Figure 2, when the user moves a GUI control, the Global Parameter changes are written using an *atomic write* operation, which only guarantees that the variable will not be half-baked after the GUI thread is halted. This is the mechanism that protects the write operation.

(2) At the very beginning of your process audio function, you access the Global Parameters and update your plugin's variables. When using Make AU in RackAFX to create your ported project, only the variables that have changed will be updated, and have *userInterfaceChange()* called and as usual, all this code is written for you.

(3) Then, you process the audio buffer as usual.

(4) Afterward, you set the global parameters that correspond the meter parameters. Again, atomic writes are used to set these variables.

(5) Lastly, the meter values are updated by read-accesses to the Global Parameters during the *idle()* loop of the GUI thread.

VST3

The VST3 API also uses an intermediate parameter container for the GUI access parameters. When using VSTGUI4 and VST3 together, there is a tight parameter binding operation that occurs which connects parameters with the same index (tagged) variables, although the code becomes somewhat complex when stepping through it (you can do this for yourself and watch the operation if you wish). The parameter container is full of Parameter objects, one for each of your plugin's GUI controls (and linked variables).

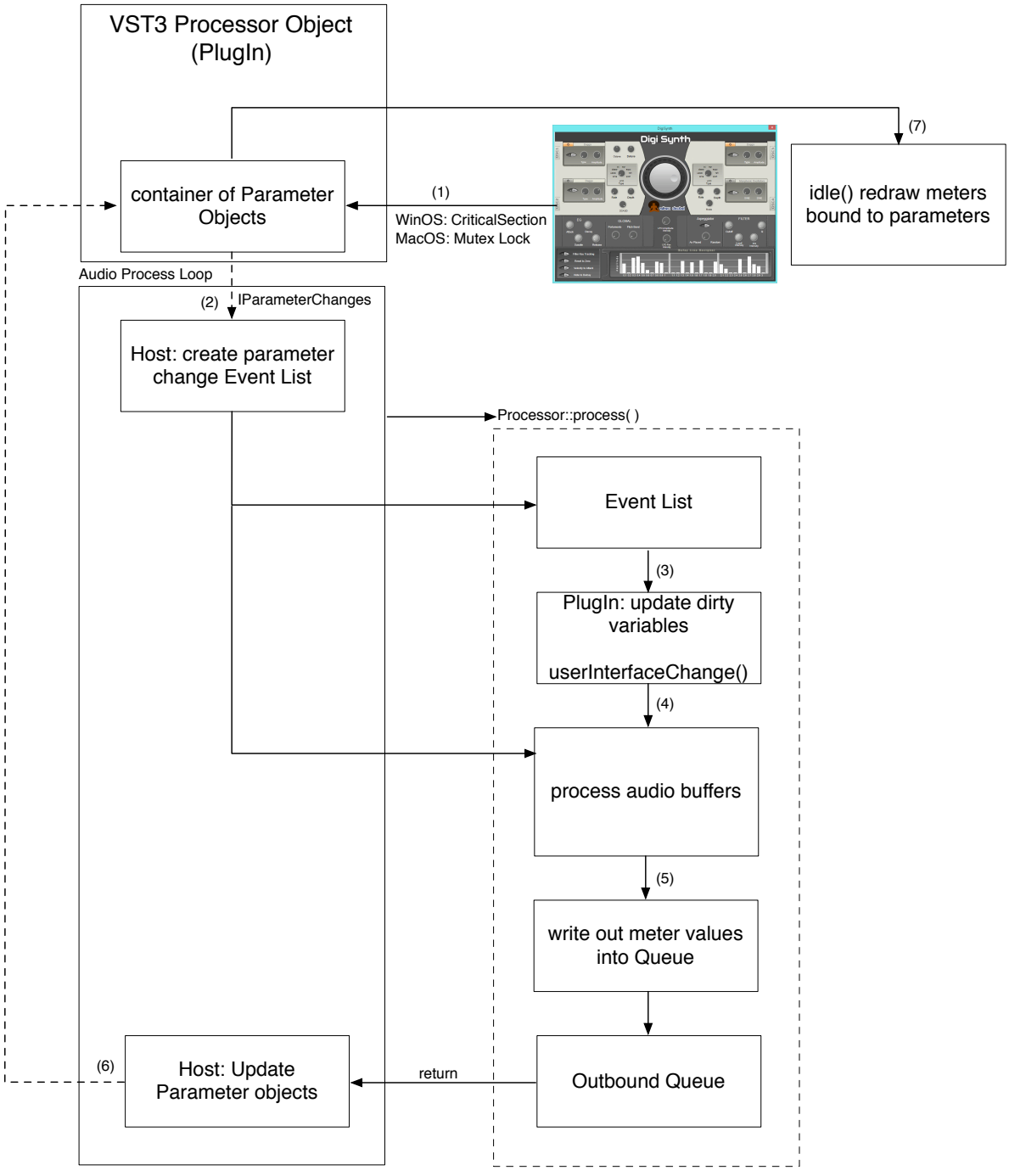

Figure 3: Audio/GUI processing in VST3

(1) Starting at the top of Figure 3, when the user moves a GUI control, the corresponding parameter object is updated in the parameter container. For WindowsOS, a CriticalSection is used while a mutex is used in MacOS during the operation. The locking occurs during the *doTriggerUpdates()* phase of operation just after the parameter value has actually been changed.

(2) At the start of the processing loop, the host creates an event list of parameter changes for any parameters that were altered and marked dirty during the previous buffer cycle. The *IParameterChanges* interface is used, though details of this are up to the authors of the VST3 host. This is shown with a dotted line because the implementation and thread safety details are handled by the host implementation.

(3) At the beginning of the *Process()* function, you iterate through the parameter change event list and alter your plugin's underlying variables, calling *userInterfaceChange()* accordingly. This can be done on a buffer-wise basis (easy) or on a sample-by-sample basis (complex).

(4) With your plugin variables set, you then process audio as normal.

(5) Meter variables are written into a special queue that is supplied as part of the argument to the *Process()* function.

(6) The Host then transfers the updated parameters into the parameter container, the details of which are left to the host authors (thus the dotted line), including thread safety and synchronization; then the outbound parameter queue is cleared.

(7) Lastly, the meter values are updated during the *idle()* loop of the GUI thread - owing to the parameter binding operation, there is no code to actually write here on the GUI side.

AAX

The AAX version is the most complex and I've very much simplified operation for this discussion. In addition, there are multiple ways of writing an AAX plugin and here I am using the "monolithic object" approach. As with the other APIs, there is a set of parameter objects (here it is a std::set) and the GUI reads and writes variables from and to these parameters. There is one parameter object for each of your plug-in's GUI controls (and linked variables). A message queue is used to queue up parameter changes, the details, which are not trivial, can be found in the AAX ported project code.

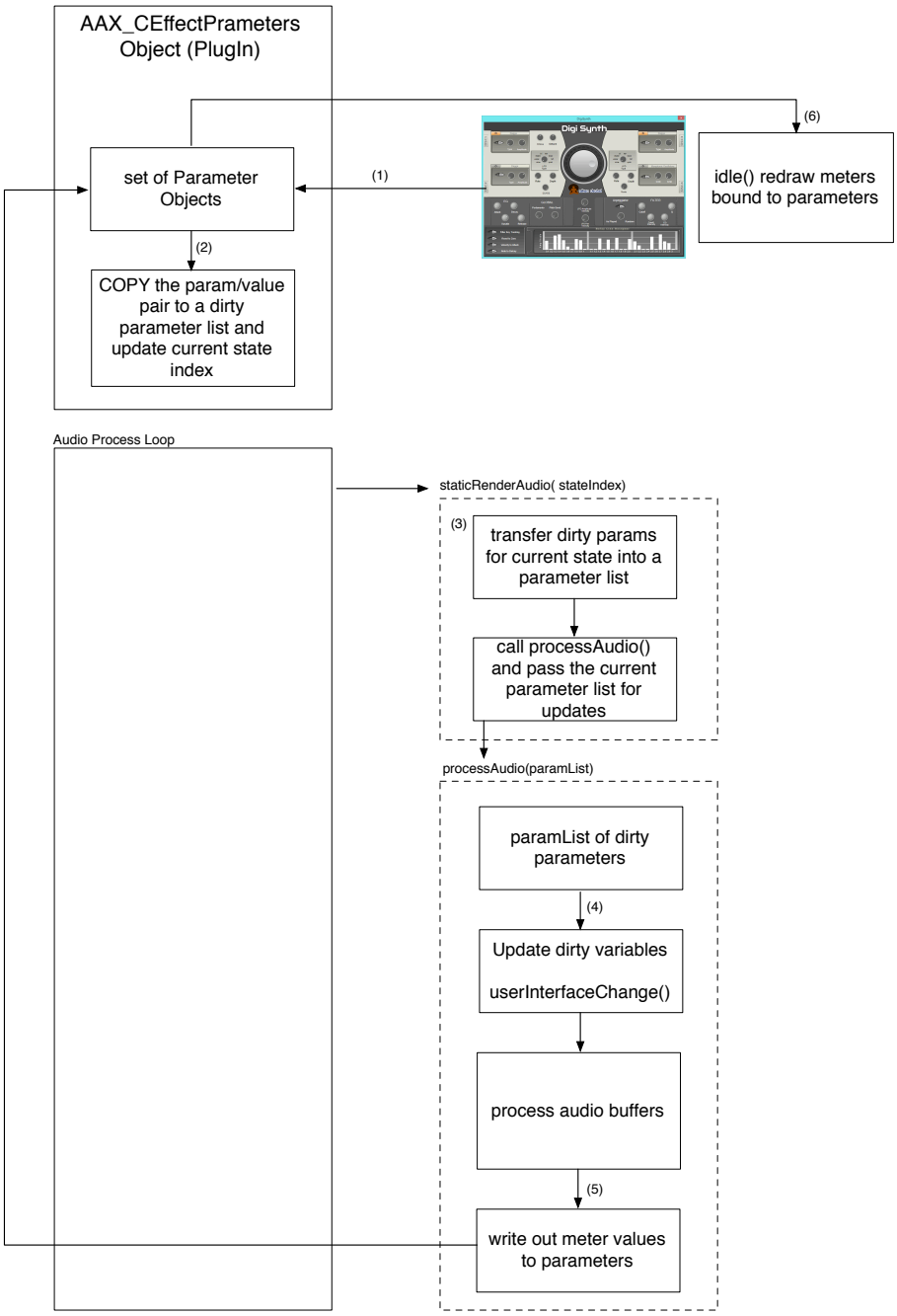

Figure 4: Audio/GUI processing in AAX

(1) Starting at the top of Figure 4, when the user moves a GUI control, the parameter object is updated.

(2) A copy of the updated parameter is added to a dirty parameter list. A mutex lock is involved during this operation but that is outside the scope of this document.

(3) Using the monolithic object approach, the audio processing is broken into two phases (and functions). During the first phase, the dirty parameter copies are transferred to a parameter list which is passed as an argument to the second-phase function.

(4) During the second phase, you alter your plugin's underlying variables according to the parameter list, calling *userInterfaceChange()* accordingly. Then, you process the audio as usual. There is also an operation to clear out the dirty parameter list after audio has been processed, which you must supply.

(5) The GUI meter variables are written into the set of parameters.

(6) Lastly, the meter values are updated during the *idle()* loop of the GUI thread.

AAX NOTE: the AAX API is a confidential and private API, however I have permission from Avid to implement the Make AAX function in RackAFX and generate projects that use this paradigm. The above information about the flow of control can be gleaned from the resulting AAX projects, even if you do not own the AAX SDK.

Final Remarks

As you can see from the above figures and explanations, all the APIs share a similar basic pattern of operation: the GUI reads and writes variable from and to some sort of intermediate container of parameters. The altered (dirty) parameter changes are ultimately applied on the plugin during the audio processing phase **prior** to signal processing, but are kept separate from the actual parameters by the use of copied arrays, copied event lists, copied parameter lists, or in the case of AU by atomic data accesses. Outbound parameter changes (meters) are then queued up at the end of the processing cycle and are ultimately applied in a thread safe manner to the GUI.