by: Devon Fritz

03/24/2017

Design Decisions

Architecture

- Actor Framework. I decided to use NI’s actor framework instead of using potentially a more fully fledged Actor Framework (ie FTW, MGI, etc…). This is due to familiarity with NI’s framework. Although I do recognize pitfalls and shortcomings of it. Below are reasons for AF:

- Extremely scalable (if this were to become something like Spotify I would be much more comfortable starting with AF)

- Forces code organization (through messages and Do.vi methods)

- Demonstrates LV ability to Tesla…at least more so than a basic QMH ☺

-

- NO zero-coupling. I decided against using zero-coupling messaging from nested actor to caller actor for "core" actors. This is due to the nature of these actors as they will are integral to the application and giving them this feature is excessive code (AKA bloat). Less is more. There may still be zero-coupled messages for abstraction layers (which will be present for peripheral features/actors).

- Not .NET. Since this application is intensely using UI elements (without LabVIEW using LabVIEWs UI strength) it would be better to use a .NET UI. Since this is a test of my LabVIEW ability, however, I decided to do the entire application in the LabVIEW environment.

- No LabVIEW trees. LabVIEW trees are difficult to format nicely and although can make drag and drop a reality are cumbersome to work with. I do not have enough time to incorporate LabVIEW trees in this application.

-No Parent Library Class. I decided against a high-level library class as I wanted to minimize the number of classes in this project.

- No uniform hidden UI controls. Normally I would standardize on using system controls and classic controls and indicators for hidden VI FPs, but due to the time intensive nature of this application I made it a non-concern.

- Use of Factory Pattern for Music Libraries. Since SPX is 1) not a real music file type and 2) most likely not the only music file type I will support I use the factory pattern to dynamically support future file types. This is the equivalence of a music codec.

- No abstraction for config data. I chose not to abstract file type for config data. This is a “personality” of the application – which basically means I do not have enough time to create an abstraction set. This is also the reason a class was not used.

- Not RT Compatible. Too much work to validate. I know for sure the xml tools are not RT compatible.

- Assume libraries will always be imported/loaded as album libraries.

- Minimize routed messages. I passed the song player enqueuer to the browser actor to avoid spending time creating the pass-through messages. I would need to consider this more on whether this is OK design or not.

- Song by reference/UID. The “raw” song data resides in the Song Manager. This means the manager is also a server of sorts that holds the loaded song data. This is done so that if the songs need to be loaded dynamically the manager can be responsible for doing so.

- XControl Bug – Xcontrols glitch updates through to front layer. Therefore I needed to work around and remove xcontrols when they are not in front and being updated

- Error Handling – There is not explicit custom error handling built in other than basic error checks and reporting in a few places. This was due to the focus on UI design as opposed to error handling architecture.

- Logging – Normally I would include logging to help debug issues and log errors. Again I did not have time to implement this as the primary focus was on UI.

UI Elements

* Font: Consolas
* Colors
  + Text Dark: 200, 200, 200
  + Background Dark: 21, 23, 26
  + Background Light: