An Introduction to the OMCF Architecture

Hewlett-Packard Development Company, L.P.

29 September 2008

1 License

Ookala Modular Calibration Framework

2 Introduction

The Ookala Modular Calibration Framework (OMCF) is an open color-calibration platform designed to support extensible, standardized color cabibration of color-critical display environments such as HP DreamColor Professional Displays.

The following document is a work-in-progress description of the original plugin architecture for `libookala` that manages the plugin registry, dictionaries, plugin chains, error handling, and color sensor plugins. Plugin loading, queries, chain execution, grouping, and data archiving are also covered.

The current OMCF package offers the `ucal` application and a combination of `hpdc_util` and `read_sensor` applets to further demonstrate this functionality.

3 Plugin Registration

Essentially, we are building a plugin registration and query system - so it makes sense to touch on the plugin registration process for a moment. Currently, registration is handled by the `PluginRegistry` class. This class maintains a list of plugin dsos, as well as object creation and deletion function pointers.

Currently, the plugin registry creates one instance of each plugin object. This makes book-keeping much easier, as other plugins can query by name for a particular object. This does not, however, mean that plugins are singletons. Its entirely possible to maintain multiple plugin registrys, each holding one distinct instance of each registered plugin object. Forcing the plugins to be singletons at this point is overly oppressive for potential future expansion.

Plugin loading is handled in `PluginRegistry::registerPlugins()`. This currently just takes a path to a module. A useful extension would be a 'load everything from this path' method, with some logic for re-loading to satisfy ordering constraints.

3.1 Steps for making a loadable plugin

As one might expect, to make a loadable plugin, we must create a class derived from `Plugin`. The `Plugin` class contains no pure virtual methods, so if you are deriving straight from `Plugin`, there should not be an necessary methods to implement (however, other "base" classes may contain pure-virtual methods).

In order to deal with symbol searching and name mangling in dynamic libraries, we need to jump through a few hoops. First, for each plugin class declaration, we need to use the macro `PLUGIN_ALLOC_FUNCS(className)`:

```
class foo: public Plugin
{
    public:
       foo();
       virtual ~foo();

       PLUGIN_ALLOC_FUNCS(foo)
       ...
};
```

Next, for each dynamic library, we need to provide a list of the plugins contained within, that we wish to register. This goes somewhere around the definition of the plugins:

```
BEGIN_PLUGIN_REGISTER(numPlugins)
  PLUGIN_REGISTER(idx, pluginName)
END_PLUGIN_REGISTER
```

This allows us to put mutiple plugins into a single dynamic library, for example:

```
class foo: public Plugin
{
    public:
        foo() {
            mName.assign("foo");
        }
```

```
            virtual ~foo() {}

            PLUGIN_ALLOC_FUNCS(foo)

        protected:
            virtual bool _postLoad();
            virtual bool _checkDeps();
    };
    class bar: public Plugin
    {
        public:
            bar() {
                mName.assign("bar");

                mAttributes.push_back("tonic");
                mAttributes.push_back("olives");
            }

            virtual ~bar() {}

            PLUGIN_ALLOC_FUNCS(bar)
    };
    BEGIN_PLUGIN_REGISTER(2)
      PLUGIN_REGISTER(0, foo)
      PLUGIN_REGISTER(1, bar)
    END_PLUGIN_REGISTER
```

Which, when compiled into a dynamic library and loaded in `PluginRegistry` will provide two plugins, `foo` and `bar`.

3.2 Plugin Queries

The `PluginRegistry` class also provides a mechanism for querying plugins. Each plugin class has a name and a list of attributes. We can query either by name, a single attribute, or a list of attributes, and return a matching plugin. If multiple plugins are found to match, we will get one that satisfies our needs:

```
std::vector<Plugin *> PluginRegistry::queryByName(const std::string name)

std::vector<Plugin *> PluginRegistry::queryByAttribute(
             const std::string attrib)

std::vector<Plugin *> PluginRegistry::queryByAttributes(
             const std::vector<std::string> attribs)
```

Plugin names are assumed to never collide, so only multiple-matching queries by attributes are supported.

The name and attributes of a plugin should be set in the constructor, by filling in strings containing the name and supported attributes of the plugin in `Plugin::setName()` and `Plugin::addAttribute()` .

3.3 Accessing the Plugin Registry from a Plugin

Often, plugins will wish to make use of the the query methods in `PluginRegistry`. To enable this, `Plugin` holds a pointer to the `PluginRegistry` that holds the instance of the plugin.

The `PluginRegistry` pointer is assigned during the plugin loading and registration phase, after `Plugin::postLoad()` executes.

3.4 Extending Plugin Methods

All the methods on `Plugin` of interest follow this convention; The outward-facing method looks normal, while the "guts" of the method fall in the _-prefixed version. This is done in cases where we need to control access to the "guts" or wrap them with more complicated logic.

Generally, if you're creating derived plugin classes, you only should have to worry about the _-prefixed versions of these functions for implementing the bulk of the work.

3.4.1 Load-time methods

We have three methods that are related to plugin loading and inter-plugin dependency checking. They will be called in the `PluginRegistry` class, after loading the dynamic library containing the plugins. Returning `false` to `PluginRegistry` is a signal that the plugin should be unloaded and not accessible.

```
virtual bool Plugin::postLoad()
virtual bool Plugin::_postLoad()
```

After being loaded, a plugin should be given a chance to fail. This would be useful for cases where the plugin supports a particular device, but that device does not exist on the system. If this fails (returns `false`), the plugin should be removed from the registry so it cannot be queried and used elsewhere.

`postLoad()` is called from within `PluginRegistry`, and is generally not something you should ever have to call.

`postLoad()` is called *before* `Plugin::mRegistry` is assigned. As such, you should never attempt to access any other plugins in `postLoad()` or `_postLoad()` This is enforced because the plugin you want may not be loaded yet. If you need to check for other plugins, you should do so in `checkDeps()`.

```
virtual bool Plugin::checkDeps()
virtual bool Plugin::_checkDeps()
```

After loading all the plugins, and making sure that they really can work to some minimal degree, we should make sure that any plugins that we rely on exist and are functioning properly. This will be called from within `PluginRegistry`, after testing `postLoad()` on all plugins.

If we return `false` to a `PluginRegistry`, the plugin will be removed. For any plugins that we require to successfully run, we need to query the plugin registry (`Plugin::mRegistry`) and call `checkDeps()` on

the returned plugin. Note that since this process can recurse, we should take care not to include anything that would cause problems when run recursivly.

```
virtual bool Plugin::postCheckDeps()
virtual bool Plugin::_postCheckDeps()
```

The final method in the plugin loading routine. It will be run on every plugin loaded in `PluginRegistry`.

This can be handy for breaking cyclical dependencies that would be introduced if all checking for other plugin support was performed in `checkDeps()`. However, it should not be though of as a replacement for `checkDeps()` since it is not ment to recursely execute on dependencies.

3.5 Example

Building upon our example declaration of two plugin classes in a dso, `foo` and `bar`, we will build up a few simple load-time checking functions.

```
// _postLoad will check that the necessary doomsday dev
// is available. If we can't destroy the whole world, there
// is no use continuing.
bool foo::_postLoad()
{
    int      fd;
    DictItem item;

    fd = open("/dev/doomsday", O_RDWR);
    if (fd < 0) {
        perror("open():");
        return false;
    }
    close(fd);
    return true;
}

// The foo plugin requires both the doomsday dev and the
// the bar plugin for successful execution. We've checked
// for /dev/doomsday in _postLoad(), but now we need to
// check for the bar plugin
bool foo::_checkDeps()
{
    std::vector<Plugin *> plugins;
    DictItem item;

    // See if we can locate the plugin in the registry.
    plugins = mRegistry->queryByName("bar");
    if (plugins.empty()) {
        fprintf(stderr,
            "foo::_checkDeps() ERROR: Can't locate plugin bar\n");
        return false;
    }
```

```
    // If it exists, we need to make sure that it also has its
    // needs filled.
    if (plugins[0]->checkDeps() == false) {
        fprintf(stderr,
            "foo::_checkDeps() ERROR: Plugin bar failed checkDeps()\n");
        return false;
    }

    return true;
}
```

4 Dictionaries

Data storage is another topic that we should touch upon. We may need a global "configuration" data space, that controls options for the entire program. Also, individual plugins may have configuration data that changes their behavior. Plugins may also need to record some data for use by other plugins.

Included is a dictionary class to satisfy these storage needs. Each entry has a name string and a data object. Dictionaries end up being used in a variety of places. First, `PluginRegistry` holds a global dictionary that each plugin contained within can access. This is useful for global options, e.g. "debug=true". Further, each `PluginChain` holds a dictionary for per-chain configuration data (See Section **??**).

Perhaps it is best to look at an example of the dictionary usage at this point:

```
Dict          d;
BoolDictItem *boolItem = new BoolDictItem;

boolItem->set(true);
d.set("debug", item);
```

Here, we see the separation between dictionary and data object. Data objects (`DictItem`) are a bit of a tricky case. We could try and make them a variant class or a union, but that makes it difficult to extend. Further, there's an issue of ownership in the above example - who is supposed to free the DictItems?

First, we'll deal with the ownership issue. Currently, allocation and deallocation of `DictItem` classes should be done through the `PluginRegistry`. Why there - because plugins may extend the `DictItem` class. This requires the allocation to be done in the plugins, and since the
PluginRegistry is the only one to know about plugin entry points, that's where the item allocation ends up.

To be somewhat more correct, the previous example should look something more like:

```
void foo(PluginRegistry *registry)
{
  Dict          d(registry);
  BoolDictItem *boolItem;

  if (registry == NULL) return;

  boolItem = registry->createDictItem("bool");

  boolItem->set(true);
  d.set("debug", item);
}
```

Here, we're allocating things appropriately. Also in this example, we need to inform a `Dict` about the registry, so it can deallocate items that it's holding when the `Dict` is destroyed.

Currently, there is a small list of built-in data types:

Argument	Data type
"bool"	BoolDictItem
"int"	IntDictItem
"double"	DoubleDictItem
"string"	StringDictItem
"blob"	BlobDictItem
"intArray"	IntArrayDictItem
"doubleArray"	DoubleArrayDictItem
"stringArray"	StringArrayDictItem

4.1 Extending DictItem

Plugins can create their own classes derived from DictItem and expose them for others to use. This can be done in a similar manner to plugin registration:

```
class MyDictItem: public DictItem
{
   ...
};

class foo: public Plugin
{
  foo();
  virtual ~foo();

  // NOTE: NO USE OF PLUGIN_ALLOC_FUNCS!

  BEGIN_DICTITEM_ALLOC_FUNCS(foo)
    DICTITEM_ALLOC("cool_item", MyDictItem)
  END_DICTITEM_ALLOC_FUNCS
  BEGIN_DICTITEM_DELETE_FUNCS
    DICTITEM_DELETE("cool_item")
  END_DICTITEM_DELETE_FUNCS
}
```

Here, "cool_item" is the string to passed into PluginRegistry::createDictItem and is also the string returned by MyDictItem::itemType().

4.2 Saving and Loading

Both the Dict and DictItem classes have serialization and unserialization methods that can be used to load and save data. In both cases, they make use of the libxml2. Data will be stored in a form:

```
<dict name="Test Dict 1">
    <dictitem type="bool" name="Debug">true</dictitem>

    <dictitem type="double vector" name="measured white Yxy">
        <value>80.012</value>
```

```
        <value>0.3333</value>
        <value>0.3333</value>
    </dictitem>
</dict>
```

Saving a `Dict` is fairly straightfoward, we just need the root node where we would like to insert the `Dict`. For example:

4.3 Managing a `Dict`

It would be nice if we were able to share `Dict` data with everyone. This way, plugins could communicate with one another, or with the outside world, by exchanging `Dict`s. This is the motivation of the `DictHash` plugin. It is a built-in plugin which should hold all the `Dict` objects that are created. It has a fairly simple interface:

```
Dict * DictHash::newDict(std::string name);
```

This will create a new `Dict` with the given name. If something already exists with this name, don't remove it, but rather just return a pointer. If we forget to name multiple `Dict`s, we don't want to loose data by over-writing.

```
bool DictHash::clearDict(std::string name);
```

Remove a `Dict` from our storage.

```
Dict * DictHash::getDict(std::string name);
```

Lookup a stored `Dict`. Returns NULL if we don't have anything stored by that name. You shouldn't hold the pointer that is returned, but instead re-query every time you need the reference within reason.

```
std::vector<std::string> DictHash::getDictNames();
```

Return a vector of all the names of `Dict`s that we hold.

5 Plugin Chains

In general, we can divide common plugins into two groups - utilities and runnables. There isn't an explicit divide between the types of plugins, its more implicit based on which methods they override. In fact, a single plugin could fall into both categories.

Utility plugins provide some service and can be used by a variety of other plugins. Examples of utility plugins include things like colorimeters or a DDC/CI connection.

Runnable plugins are those that "do" something, and can be grouped together into a chain with other plugins to run some task. The chain is then executed serially. Runnable plugins will make use of utility plugins, but generally not the other way around. An example of a runnable plugin would be a calibration algorithm.

For these runnable plugins, we have methods for dealing with the chain of execution. Utility plugins will probably not need to override these methods.

5.1 Grouping Runnable Plugins into Chains

The `PluginChain` class provides a mechanism for grouping together runnable plugins into something useful. Each plugin chain has facilities for building a chain of plugins, accessing the plugins in the chain, and storing dictionary data for the plugins to use.

In general, you should not have to worry much about the details of building a `PluginChain`, though you will likely make great use of its helper functionality.

Running a `PluginChain` consists of a few passes over the list of plugins. First, the `preRun()` method is called on each plugin. If this succeeds, we proceed to the next step of calling `run()` on each plugin. Finally, `postRun()` is called. `postRun()` will always be called, regardless of the success of the `preRun()` and `run()` steps.

5.2 Runnable Plugin Methods

Each of these methods has access to the `PluginChain` from which it is being executed. This affords the plugins two useful pieces: querying other plugins in the chain and canceling the execution of the chain.

Querying other plugins is especially useful for finding a gui plugin to use. Gui plugins should be added to chains, as they are runnable (generally showing and hiding the display in `preRun()` and `postRun()` respectively). The alternative would be to query the `PluginRegistry` to find a gui plugin. However, there could potentially be multiple gui-providing plugins loaded at one time, and we would need some way of arbitration. Instead, we include gui plugins in the chain, only including one gui plugin per chain. This way multiple chains can be constructed that have differing guis.

Plugins in the chain have two methods available through the `PluginChain` dealing with canceling the execution of the chain. First, `PluginChain::cancel()` will set the cancel flag and stop execution of further plugins in the chain. As a plugin writer, you should call this if something fatal happens or if the user sends a cancel signal. Also, since plugins may yield to the main gui loop to allow for handling events (e.g. cancel-button clicks), `PluginChain::wasCancelled()` should be tested occasionally to see if processing should be aborted.

```
virtual bool Plugin::preRun(PluginChain *chain)
virtual bool Plugin::_preRun(PluginChain *chain)
```

Prior to running a chain, we should give nodes the opportunity to do any per-run things. Here we might put things like showing a gui, connecting to gui signals, and so on.

`preRun()` will be called when a plugin chain is run. Returning false will skip `preRun()` for the remaining plugins in the chain, skip `run()` for all plugins in the chain, but execute `postRun()` for all plugins.

```
virtual bool Plugin::run(PluginChain *chain)
virtual bool Plugin::_run(PluginChain *chain)
```

This would be the place to put the guts of what you want to do. It will probably query for other plugins, either from the registry or from the chain.

Returning `false` will skip `run()` for the remaining plugins in the chain but still call postRun() on all plugins.

```
virtual bool Plugin::postRun(PluginChain *chain)
virtual bool Plugin::_postRun(PluginChain *chain)
```

After a chain is run, we should give each plugin a chance to cleanup anyhthing that it might have setup.

5.3 Example Chain Functionality

As an example of what the various run-time methods may implement, we outline a simple calibration process detailing what might occur in each method.

preRun:	Gui opens
	Colorimeter setup
	Colorimeter init
run:	Reset LUTs
	Calibration loop - display colors and measure
	Save calibration data
	Update LUTs
postRun:	Colorimeter shutdown
	Gui closes
	If canceled, reset state

5.4 Queries for Plugins in a Chain

Similar to the plugin query functionality in `PluginRegistry`, we can also query plugins that live in a particular chain. This can be useful if you wish to specify which plugins are used by other plugins, instead of relying on whatever we find in the `PluginRegistry`.

```
Plugin *     PluginChain::queryByName(const std::string name);
Plugin *     PluginChain::queryByAttribute(const std::string attrib);
Plugin *     PluginChain::queryByAttributes(
                          const std::vector<std::string> attribs);

bool PluginChain::queryByAttribute(const std::string attrib,
                        std::vector<Plugin *>&matches);
bool PluginChain::queryByAttributes(
                        const std::vector<std::string> attribs,
                              std::vector<Plugin *>&matches);
```

5.5 Plugin Chain Gang

While chains of plugins are useful, we also need a way to keep track of all the chains, much the way plugins are tracked by `PluginRegistry`. For this, we have the `PluginChainGang` class.

`PluginChainGang` is derived from `Plugin`, so there should be only one instance per `PluginRegistry`, and it is queryable through the `PluginRegistry`.

On load, `PluginChainGang` loads up chain configurations specified in a file. On disk, chains are stored something like:

```
<chain>
    <name>The Name Of the Chain</name>

    <plugin>Plugin 0</plugin>
    <plugin>Plugin 1</plugin>
    <plugin>Plugin 2</plugin>

    <dict>
        <dictitem type="bool">
            <name>option 0</name>
            <value>true</value>
        </dictitem>
        ...
    </dict>
</chain>
```

The list of plugins, in order of execution can be specified, as well as as the configuration dictionary for the chain. Chains for which all plugins were loaded successfully are stored. Query methods are available to see the names of available chains. Run methods are available for starting execution of a particular chain, and a cancel method is available for externally canceling the currently running chain.

5.5.1 Periodic Chain Execution

It is occasionally hand to have chains execution continuously at some specified interval. For example, implementing a status monitor that polls the current calibration state could be done this way.

To make a chain execution periodically, add a double value to the chain dictionary named "period". The value should be the execution interval in seconds. Note that if another chain is executing when the timer expires, the period chain may not run.

XXX: We still need to figure out where to specify the file that this data should be pulled from. Currently, it gets pulled from the string attribute "Chain Config Files" in the global dictionary.

XXX: We also need to add external chain addition methods.

XXX: It would be handy to include chains that are run when the program kicks off

6 Error Handling

`Plugin` contains an error string that plugin implementors should set before returning `false` in error cases.

The error string can be retrieved using `Plugin::errorString`.

7 Color Sensor Plugins

Simliar to the motivation for a base class for gui's, we also have a base class for color probes. This way, a plugin that needs color measurement capabilities can do so independently of which probes are attached (and have appropriate plugins loaded).

7.1 Sensor

Sensors may be spectrometers or colorimeters, or things of that nature. We've still named the base class `Sensor`, for ease of typing (But if you have a better name - speak now!).

`Sensor` has three pure-virtual methods: `_init()`, `_shutdown()`, and `_measureYxy()`

```
virtual bool Sensor::init()
virtual bool Sensor::_init() = 0;
```

Most devices require some sort of one-time setup. We'll setup things such that if this hasn't happened by the time we make our first measurement, we'll run `init()`.

`init()` is different from `postLoad()`. `postLoad()` should just test for the existence of a device, and fail if one is not found. We need sometime between `postLoad()` and the first time we measure where we can config the device. This could be done in `postCheckDeps()`, but we still need to allow for some configuration before starting up the probe.

```
virtual bool Sensor::shutdown()
virtual bool Sensor::_shutdown() = 0;
```

Does any one-time shutdown procedures needed to cleanup after the device.

```
virtual bool Sensor::measureYxy(double *valueYxy)
virtual bool Sensor::_measureYxy(double *valueYxy) = 0;
```

Sample from a device to read a value. Results should come back as Yxy in the desired units.

Its a good practice to call `init()` before `measureYxy()`, as `init()` may need to do its own measuring or manipulating of the display output. Failing to do so may be an error.

The version you want to implement is `_measureYxy()`.

`measureYxy()` assumes that your `_measureYxy()` will block while it is reading. So, we'll start up a thread in `measureYxy()` to call `_measureYxy()` and handle main loop events while we wait.

```
virtual bool Sensor::setUnits(const std::string units)
const std::string Sensor::getUnits()
```

Set the desired measurement units. This should be done prior to `init()`. Common options will include:

- "fL"
- "cd/mˆ 2" or "nits"

The parameters are strings to avoid enum fighting from people wanting to use non-standard values. This does mean that you should be somewhat careful to check return values.

```
virtual bool Sensor::setDisplayTech(const std::string display)
const std::string Sensor::getDisplayTech()
```

Set the display technology. This should be done prior to `init()`. Common options will include things like:

- "crt"
- "lcd"
- "led crt"
- "dcinema dlp"

The parameters are strings to avoid enum fighting from people wanting to use non-standard values. This does mean that you should be somewhat careful to check return values.

```
bool Sensor::setIntegrationTime(const std::string length);
const std::string Sensor::getIntegrationTime();
```

Set the integration time, based on a category. If you have setup to measure a CRT and choose AUTO mode here, you should be prepared for the colorimeter to read the screen for refersh rates during `init()`, which may require cranking the luminance output of the display (which you should do prior to calling `init()` since the colorimeter doesn't really know which display you are using). Common options will include:

- "auto"
- "fast"
- "slow"

The parameters are strings to avoid enum fighting from people wanting to use non-standard values. This does mean that you should be somewhat careful to check return values.

8 Saving Data

Thus far, we have touched on configuration files for plugin chains. These files are read-only, as far as the application is concerned. This then begs the question of how to persist data to disk. There are a variety of potentially interesting pieces to persist, such as the time of last calibration, display readings used to compute lookup tables, and so forth.

Enter the `DataSavior` plugin. This is just a dictionary with some associated file on disk that can be loaded and saved. The dictionary can be accessed through `DataSavior::mDict`.

The on-disk files to use are configured through a vector of strings in the global dictionary named "DataSavior Files". File names can contain a **%s** which is substituted for the current host name. Multiple files can be specified.

On `DataSavior::load()` and in `DataSavior::_postLoad()`, the contents of the *newest* data file are read into the dictionary.

On `DataSavior::save()`, the contents of the dictionary are saved to *all* data files.

This strategy was chosen to allow for storing data in multiple places for redundancy, should one of the locations not be available. For example, one file could live on a networked file system while another lives on the local file system. Should the network be unavailable, we should still be able to save locally. Then, when loading, if the network was available, the local file would be chosen as it has a later modification time.