Kickstart: New HMI Framework

How to use the new HMI Framework architecture

Version 0.5

November 2017

Abstract

This document presents the new HMI Framework[1], and explain the basics steps to migrate an existing AGL app to use this new architecture.

Table of contents

Document revisions

Date	Version	Designation	Author
31 Oct. 2017	0.1	Initial release	R. Forlot
02 Nov. 2017	0.2	Change Legals Copyright	R. Forlot
07 Nov. 2017	0.3	Change details about GUI libraries	R. Forlot
13 Nov. 2017	0.4	GUI libraries now moved in their repo	R. Forlot
14 Nov. 2017	0.5	Added soundmanager instructions	K. Mitsunari

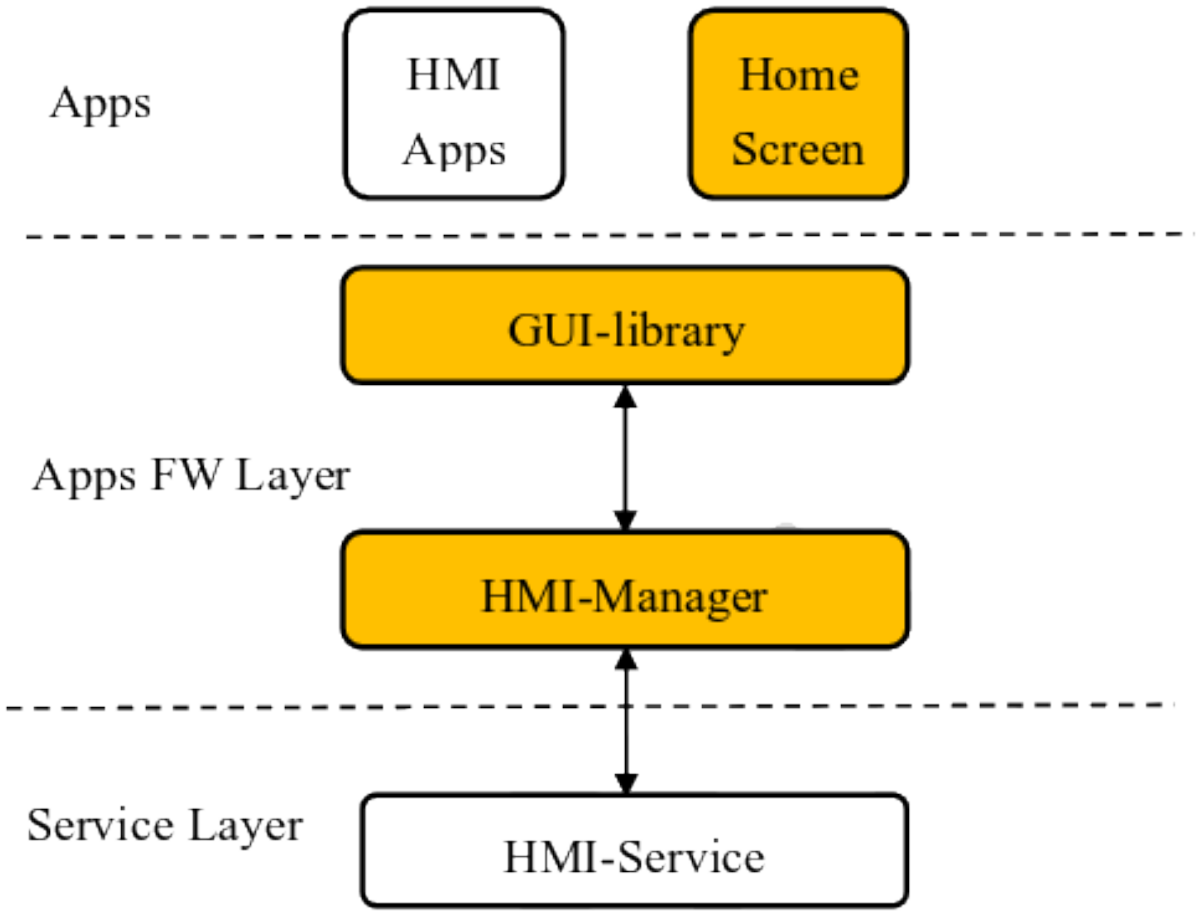
Legals

1 https://wiki.automotivelinux.org/_media/eg-ui-graphics/170802_agl_hmi-fw_arch_0_2_4.pdf

1. Presentation

1.1. Global architecture

Here are some short explanation of the new HMI Framework Architecture and better than a long speech:



HMI Framework Architecture

HMI Service will be composed with AGL Service, 3 at time of writing :

- AGL Service WindowManager[2]
- AGL Service HomeScreen[3]

2 https://gerrit.automotivelinux.org/gerrit/#/admin/projects/apps/agl-service-windowmanager-2017
3 https://gerrit.automotivelinux.org/gerrit/#/admin/projects/apps/agl-service-homescreen-

- AGL Service SoundManager[4]

HMI Manager layer is a set a corresponding library used to call API implemented by HMI Service, in addition of others functions that an OEM or Vendor would want to implement.

GUI library is the GUI tool kit itself such like Qt, HTML5 and so on but apps can access to the HMI Manager directly using websocket. You could find these helper libraries in their own repositories in gerrit[5][6][7].

1.2. HMI components general principle

HMI is composed of 3 components : Homescreen, WindowManager and SoundManager. Each one use the same principle of event handler which you could associate to a callbacks.

Callbacks mainly will launch actions to get something done (register an app, surface to draw, play a music stream, make invisible a surface, …).

WindowManager understand about now, these events:

- Event_Active
- Event_Inactive
- Event_Visible
- Event_Invisible
- Event_SyncDraw
- Event_FlushDraw

Homescreen :

- Event_TapShortcut
- Event_OnScreenMessage
- Event_OnScreenReply

Soundmanager got the event handler directly inside its library and service.

2017
4 https://gerrit.automotivelinux.org/gerrit/#/admin/projects/apps/agl-service-soundmanager-2017
5 https://gerrit.automotivelinux.org/gerrit/#/admin/projects/staging/qlibsoundmanager
6 https://gerrit.automotivelinux.org/gerrit/#/admin/projects/staging/qlibhomescreen
7 https://gerrit.automotivelinux.org/gerrit/#/admin/projects/staging/qlibwindowmanager

2. How-to adapts HVAC to use new HMI Framework

2.1. Clean old HMI usage code

HVAC use the old homescreen architecture and library, this has to been wiped before going further.

We will use the new libhomescreen and header file is the same, inclusion will remains but all related code will be removed from the app. Here from app/main.cpp

```
-#ifdef HAVE_LIBHOMESCREEN
 #include <libhomescreen.hpp>
-#endif

 int main(int argc, char *argv[])
 {
-#ifdef HAVE_LIBHOMESCREEN
-    LibHomeScreen libHomeScreen;
-
-    if (!libHomeScreen.renderAppToAreaAllowed(0, 1)) {
-        qWarning() << "renderAppToAreaAllowed is denied";
-        return -1;
-    }
-#endif
-
```

As app name will be used several times, better to store it in a variable and use it:

```
 int main(int argc, char *argv[])
 {
+    QString myname = QString("HVAC");;

     QGuiApplication app(argc, argv);
```

2.2. Make WindowManager handle your app

New WindowManager use a configuration file describing layer mapping. By default, a sample file, layers.json is used located in AFM_APP_INSTALL_DIR, */var/local/lib/afm/application/windowmanager-service-2017/0.1* at time of writing this guide.

You could create and use your own configuration file following format described in the documentation[8] setting up the environment variable LAYERS_JSON. To use this environment variable you could add it in the windowmanager service unit file located

8 https://gerrit.automotivelinux.org/gerrit/gitweb?p=apps/agl-service-windowmanager-2017.git;a=blob;f=doc/ApplicationGuide.md;h=25d87be52f11ec007457166bfbbaf1ba39fb3f7b;hb=refs/heads/master#l193

in */usr/local/lib/systemd/user/afm-service-windowmanager-service-2017@0.1.service*

or add it in a file that you may have to create */var/run/afm-debug/windowmanager-service-2017@0.1.env.*

Sample file known about default demo AGL app only. If you need to add yours, you have to edit it. Following, an example in green:

```
"mappings": [
  {
     "role": "^HomeScreen$",
     "name": "HomeScreen",
     "layer_id": 1000,
     "area": { "type": "full" },
     "comment": "Single layer map for the HomeScreen, XXX: type is redundant,
could also check existence of id/first_id+last_id"
  },
  {
     "role": "MediaPlayer|Radio|Phone|Navigation|HVAC|Settings|Dashboard|POI|
Mixer|YourApp",
     "name": "apps",
     "layer_id": 1001,
     "area": { "type": "rect", "rect": { "x": 0, "y": 218, "width": -1,
"height": -433 } },
     "comment": "Range of IDs that will always be placed on layer 1001,
negative rect values are interpreted as output_size.dimension - $value",

     "split_layouts": [
        {
           "name": "Navigation",
           "main_match": "Navigation",
           "sub_match": "HVAC|MediaPlayer",
           "priority": 1000
        }
     ]
  },
  {
     "role": "^OnScreen.*",
     "name": "popups",
     "layer_id": 9999,
     "area": { "type": "rect", "rect": { "x": 0, "y": 760, "width": -1,
"height": 400 } },
     "comment": "Range of IDs that will always be placed on the popup layer,
that gets a very high 'dummy' id of 9999"
  }
]
```

We see that HVAC is already present, so far there is no need to change the file.

2.3. Make your app use WindowManager

As we take an existing app, WindowManager already know it, so it isn't needed to modify layers.json as see in the previous section.

This is an app using Qt, a GUI library already exist[9][10] qlibwindowmanager, add the header file to the main app file include statement as well as QQuickWindow to later use bringing the window on screen explained in §2.5 chapter :

```
#include <QtCore/QDebug>
#include <QtCore/QCommandLineParser>
#include <QtCore/QUrlQuery>
#include <QtGui/QGuiApplication>
#include <QtQml/QQmlApplicationEngine>
#include <QtQml/QQmlContext>
#include <QtQuickControls2/QQuickStyle>
+#include <qlibwindowmanager.h>
+#include <QQuickWindow>
#include <libhomescreen.hpp>
```

Then instanciate a windowmanager object, this should be provided by the GUI library.

HVAC use QML engine, so add windowmanager object once arguments process. Once you got a WindowManager you need to register your apps while requesting resources and provide the app name:

```
         bindingAddress.setQuery(query);
         QQmlContext *context = engine.rootContext();
         context->setContextProperty(QStringLiteral("bindingAddress"),
bindingAddress);
    }
+
+        std::string token = secret.toStdString();
+        QLibWindowmanager* qwm = new QlibWindowmanager();
+
+        // WindowManager initialization
+        if(qwm->init(port,secret) != 0){
+            exit(EXIT_FAILURE);
+        }
+        // Request a surface as described in layers.json windowmanager's file
+        if (qwm->requestSurface(myname)) != 0) {
+            exit(EXIT_FAILURE);
+        }
+        //  Create an event handler against an event type. Here a lambda is
called when SyncDraw event occurs
+        qwm->set_event_handler(QLibWindowmanager::Event_SyncDraw, [qwm,
myname](json_object *object) {
+            fprintf(stderr, "Surface got syncDraw!\n");
+            qwm->endDraw(myname);
+        });
+
    engine.load(QUrl(QStringLiteral("qrc:/HVAC.qml")));
```

Once surface is ready, windowmanager notify the app with a SyncDraw event, then app can began to draw on that surface its UI.

When drawing is finished app let the windowmanager know that drawing ends sending it an event enddraw. Here drawing is just about empty the surface allocated to the

9 https://gerrit.automotivelinux.org/gerrit/gitweb?p=apps/agl-service-windowmanager-2017.git;a=tree;f=client;h=d2303204f96c3d8e155cdece69939b60cf0b2f0d;hb=refs/heads/master

10 https://gerrit.automotivelinux.org/gerrit/#/admin/projects/staging/qlibwindowmanager

app. QML will later write in it using HVAC.qml resource file.

Now that we handled windowmanager and surface ready to be displayed, we need to command homescreen to handle launch of the application.

2.4. Add package config

To link to qlibwindowmanager and libhomescreen, we will add package name to the .pro file. Here from the app/app.pro

```
 TARGET = hvac
 QT = quickcontrols2

+CONFIG += link_pkgconfig
+PKGCONFIG += qlibwindowmanager libhomescreen
```

2.5. Launch application using new Homescreen

Homescreen app provided by default manages apps launch as well as displaying OnScreen message, by example a alert, and its answer.

To launch the HVAC application from new Homescreen we need to implement an event handler like we did for windowmanager with SyncDraw event:

```
+
+        // HomeScreen
+        hs->init(port, token.c_str());
+        // Set the event handler for Event_TapShortcut which will be delivered
when the user click the short cut icon.
+        hs->set_event_handler(LibHomeScreen::Event_TapShortcut, [qwm, myname]
(json_object *object){
+            json_object *appnameJ = nullptr;
+            if(json_object_object_get_ex(object, "application_name",
&appnameJ))
+            {
+                const char *appname = json_object_get_string(appnameJ);
+                if(myname == appname)
+                {
+                    qDebug("Surface %s got tapShortcut\n", appname);
+                    qwm->activateSurface(myname);
+                }
+            }
+        });
+
     engine.load(QUrl(QStringLiteral("qrc:/HVAC.qml")));
```

Here we just ensure that event coming from is indeed for us comparing the application_name with our app name. If so, application can requests windowmanager to activate the surface.

2.6. Get application displays in homescreen

An additional step must be done to actually get the app displayed in homescreen. Until then we only empty and prepare surface to get filled by QML. Which is done with:

```
engine.load(QUrl(QStringLiteral("qrc:/HVAC.qml")));
```

So, after writing image, an application requests rendering to windowmanager.

The signal "*frameSwapped*" means the end of writing application(QML) image, you just have to activate the surface in reaction to that signal to get your app correctly drawed in your homescreen:

```
          engine.load(QUrl(QStringLiteral("qrc:/HVAC.qml")));
+         QObject *root = engine.rootObjects().first();
+         QQuickWindow *window = qobject_cast<QQuickWindow *>(root);
+         QObject::connect(window, SIGNAL(frameSwapped()), qwm,
SLOT(slotActivateSurface()
+         ));
```

Here, slotActivateSurface is a function from GUI library *qlibwindowmanager.cpp.*

3. SoundManager

To show how to integrate soundmanager to your application, we will show the example to use radio application[11]. The architecture is described in here[12].

But audio architecture is under heavy discussing, so this contents **must** be modified.

3.1. Add package config

For Qt application, it is useful to use qlibsoundmanager, because we will add the package name to the .pro file. Here from the app/app.pro

```
 TARGET = radio
 QT = quickcontrols2

+CONFIG += link_pkgconfig
+PKGCONFIG += qlibsoundmanager
```

3.2. Make your application use SoundManager

To enable radio app to use soundmanager, add header file to use. Here from the app/main.cpp

```
#include <QtQml/QQmlApplicationEngine>
#include <QtQml/QQmlContext>
#include <QtQuickControls2/QquickStyle>
+ #include <qlibsoundmanager.h>
```

Then, initialize qlibsoundmanager with the port and token given by application framework. With this variable, library establishes the connection to soundmanager server.

```
query.addQueryItem(QStringLiteral("token"), secret);
bindingAddress.setQuery(query);
context->setContextProperty(QStringLiteral("bindingAddress"), bindingAddress);
+QlibSoundmanager *smw = new QlibSoundmanager();
+smw->init(port, secret);
```

11 https://gerrit.automotivelinux.org/gerrit/gitweb?p=apps/radio.git;a=summary
12 http://docs.automotivelinux.org/docs/apis_services/en/dev/reference/hmi-framework/3_3-SoundManager-Guide.html

Finally, setup receiving reply/event signal from library.

In this example, we connect reply/event signal from qlibsoundmanager library to slot function written in QML to deliver reply and the event of the sound state change.

```
+        engine.rootContext()->setContextProperty("smw",smw); // if you would
like to use library from QML
           engine.load(QUrl(QStringLiteral("qrc:/Radio.qml")));

+        QObject::connect(smw, SIGNAL(reply(QVariant)),
+            root, SLOT(slotReply(QVariant)));
+        QObject::connect(smw, SIGNAL(event(QVariant, QVariant)),
+            root, SLOT(slotEvent(QVariant, QVariant)));
```

3.3. Communicate with soundmanager

To use soundmanager, there are several phases to step.

- Register your application
- Connect your application(source) to sink
- Get approval feedback to output sound from Sound Manager, then make decision to play/stop radio.

First, application needs to register application to get sourceID. Application name shall be defined by system, then the source(application) list is defined in /etc/audiomanager/configuration.xml. The sample configuration is located in here[13]. SourceID variable is used for getting connectionID and audio policy management. SourceID is returned by reply signal. In this example, sourceID is set in QML, but of course this can be in C++ side. Here from app/Radio.qml.

```
ApplicationWindow {
    id: root
+    property int  sourceID
+    property int  connectionID
+    signal disconnected
+    signal paused
+    signal connected
+    function slotReply(msg){
+        var jstr = JSON.stringify(msg)
+        var content = JSON.parse(jstr);
+        var verb = content.response.verb
+        var err = content.response.error
+        switch(verb)
+        {
+            case "connect":
+                if(err == 0){
+                    connectionID = content.response.mainConnectionID
```

13 https://gerrit.automotivelinux.org/gerrit/gitweb?p=apps/agl-service-soundmanager-2017.git;a=tree;f=conf/audiomanager-config-sample;h=0bf3f419afa40e295487e599f6d32c3a61f42140;hb=HEAD

```
+                }
+                break;
+            case "registerSource":
+                if(err == 0){
+                    sourceID = content.response.sourceID
+                }
+            default:
+                break;
+        }
+    }
// Skipping
+        Component.onCompleted: {
+            smw.registerSource("radio")
+        }
}
```

The case of "connect" in slotReply is related to the next.

Second, in this example, we connect own to default sink when the radio application is rendered. After rendering, WindowManager will emit flushDraw signal to applications, so call connect function in that time. Of course, the timing of calling "connect" is defined by user, for example, application can call "connect" reacted by user operation such as touch operation, other incoming event and so on.

Sink means abstract output point of the audio stream. SinkID "default" is only available for now, or you can designate as number.

```
+        qwm->set_event_handler(QLibWindowmanager::Event_FlushDraw, [&engine,
smw](json_object *object) {
+            QObject *root = engine.rootObjects().first();
+            int sourceID = root->property("sourceID").toInt();
+            smw->connect(sourceID, "default");
+            });
```

After calling "connect", SoundManager will return reply with "mainConnectionID". Connection means the connection between source and sink.

Finally, let's write the getting approval function. In this example, in main.cpp, the event signal is connected to slotEvent function in QML.

```
+    function slotEvent(event,msg){
+        var jstr = JSON.stringify(msg)
+        var content = JSON.parse(jstr);
+        var eventName = content.event
+        switch(eventName)
+        {
+            case "soundmanager\/asyncSetSourceState":
+                if(sourceID == content.data.sourceID){
+                    smw.ackSetSourceState(content.data.handle, 0)
+                    switch(content.data.sourceState){
+                        case "on":
```

```
+                                connected()
+                                break;
+                            case "off":
+                                disconnected()
+                                break;
+                            case "paused":
+                                paused()
+                                break;
+                        }
+                    }
+                    break;
+                default:
+                    break;
+            }
+        }
```

The event of “asyncSetSourceState” is automatically subscribed in the library.

After calling “connect” or an other application get connected, this event will be emitted. An Application can decide the behavior in response to the connected/disconnected/paused signals.