QML Runtime Reload

Visit us at http://www.ics.com

Produced by Integrated Computer Solutions

Material based on Qt 5.5.x

Why Qt?

- Write code once to target multiple platforms
- Produce compact, high-performance applications
- Focus on innovation, not infrastructure coding
- Choose the license that fits you
  - Commercial, LGPL or GPL
- Count on professional services, support and training
- Take part in an active Qt ecosystem

20 years of customer success and community growth

About ICS

- Founded in 1987
- Trolltech/Nokia/Digia's Qt Training Partner since 2002
- Provider of integrated custom software development and user experience (UX) design
- For embedded, touchscreen, mobile and desktop applications
- HQ in Bedford, Massachusetts with offices in:
- USA: California, Canada, Europe
- 120+ employees worldwide

See http://www.ics.com/

User Experience

We blend the art of visual design and agile engineering to help our clients develop successful products. The following is a list of the core services and skills that the ICS User Experience Design team can provide to your organization.

- **Product Vision & Direction**
- **User Experience Design**
- **Usability Research & Testing**
- **Visual & Motion Design**

What is Runtime Reload?

'Runtime Reload' is simply a description for the process of refreshing QML content at runtime. It has many uses including:

- Implementing UX designs expeditiously
  - Working alongside design team in a 'Live Coding' environment
- Changing the set of qml files to another one, i.e. 'theming' or 'skinning'
- Saving time overall and providing a smoother transition for languages or visual assets for the user
- No need to re-compile to see changes.

Application Design Considerations

Reloading the current QML resources while the application is running does require some attention to application design and implementation.

- You will want to make sure you keep a pointer to your *QQuickView* or *QQmlApplicationEngine*
  - *QQuickView* and *QQmlApplicationEngine* have different approaches to reload QML.
- Ensure the application is reading from the files in your source tree, not the QRC file.
- Utilize *QFileSelector*, *QQmlFileSelector* for easier deployment of resources.
- State should be kept in c++, not QML
  - Things like the current view, relevant properties for the UI should be stored in c++ as properties the QML can simply act upon.
- Keep your path to the QML as a member variable, so that it can be changed if needed.

Application Design Considerations

- If the QML UI is too closely ‘married’ to the c++ backend, i.e. the backend has pointers to QQuickItems in the view, when you try to reload you have to keep track of those items and destroy / recreate them properly or you can potentially have invalid pointers.
- Reloading Errors - If there is an issue with the updated code, or the new theme, need to create own recovery mechanism for ideal user experience.
- Shadow builds will change the application directory path to one that is ‘outside’ your source tree.
  - This needs to be accounted for, either by using the ‘Run Settings’ tab to set a working directory or programmatically in the code itself.
- If you have custom QML plugins, ensure the lifecycle can accommodate sudden removal of the UI layer.
  - As with the QML UI being too closely connected with the c++ backend, the same should be avoided with plugins.

Reloading a QQuickView

Reloading a *QQuickView* is a fairly straight-forward process. The files will be loaded from the disk, not the QRC. This is done by setting the path to your QML files on the disk as the loading point. Two functions are responsible for making the calls to complete the reload action.

- reload() - Sets a timer connected to the actual slot where we will do the work. We found in testing that a direct function call will cause the application to crash, therefore we set a timer with a signal that will call the delayReloadQml() function.
- delayReloadQml() - This function will do two things:
  - call QQuickView->clearComponentCache() which will invalidate the current cache of QML files
  - call QQuickView->setSource() with the same path to the root QML file. This will cause the root QML element to be reloaded, as well as any child components.
- Instead of a timer, one could also utilize this method
  - ***QMetaObject::invokeMethod(this, "delayReloadQml", Qt::QueuedConnection);***

QQuickView Reloading Code

Excerpts from an Application Reload Functions

```
void AppManager::reload()
{
  qDebug() << Q_FUNC_INFO;
  QTimer *delayTimer= new QTimer(this);
  delayTimer->setInterval(500);
  delayTimer->setSingleShot(true);
  connect(delayTimer, SIGNAL(timeout()), this, SLOT(delayReloadQml())); delayTimer->start();
}

void AppManager::delayReloadQml(){
  mView->engine()->clearComponentCache();
  mView->setSource(QUrl(mSelector->select(m_qmlPath+"main.qml")));
}
```

mSelector refers to *QFileSelector* member variable

m_qmlPath refers to the qml path we are loading from

- Local path like ‘file:///path/to/qml/in/source/tree
- QRC path ‘qrc:/’
- Http path “http//192.168.80.1/qml/”

Loader Considerations with QQuickView

While *QQmlFileSelector* will automatically choose files for you, this functionality can be broken by using a QML *Loader* depending on how your application is constructed. A simple way to handle this issue is to provide access to the QFileSelector, allowing you to return the corrected path.

AppManager.h

```
Q_INVOKABLE QString adjustPath(QString qmlFile);
```

AppManager.cpp

```
QString AppManager::adjustPath(QString qmlFile)
{
  QString returnPath =  mSelector->select(m_qmlPath+qmlFile);
   return returnPath;
}
```

As you can see, the path selected via the QFileSelector is returned via this function and used like this:

```
Loader{
    id:contentLoader
    source:app.adjustPath(app.currentScreen)
    anchors{
       top:menu.bottom
       left:parent.left
       right:parent.right
       bottom:parent.bottom
    }
 }
```

In most cases this isn't needed, but just in case....

Embedded Device Strategies

Working on embedded devices can provide their own set of challenges when you wish to reload the UI to see how it performs on the device, versus a nice shiny laptop running a simulator. In these cases, there are different options available to you.

- ***NFS*** - NFS or a network share that the application can reach is a great option.
- ***Http*** - You can also utilize a web server, but it is more restrictive. You simply need to point to the network location of the QML. There are some drawbacks to this method
  - You need to create qmldir files
  - QFileSelector functionality does not work.
  - Network issues, configuration overhead to get started.
  - Should only be used in development as http is insecure without some form of additional security layer (Https vs Http)

How about QQuickWindow?

If you are utilizing a *QQuickWindow*, all hope is not lost. While the mechanism functions differently, the same result can be achieved. It is important to note some differences in *QQmlApplicationEngine* versus *QQuickView*

- The *QQmlApplicationEngine* is designed to run a full application. As a result it combines some of the characteristics of a *QQuickView* and a *QQmlEngine*
  - Notably, you utilize *QQmlApplicationEngine*::load() as opposed to the *QQuickView*::setSource() function
  - Attempting to call the load function multiple times will result in multiple windows. To avoid this, you need to close the existing *QQuickWindow* first.

To reload a *QQuickWindow*:

- Call the close() function on the window, via c++ or QML
- Call clearComponentCache() on *QQmlApplicationEngine*
- Call load() again to load the root QML file with the updated content

State of The Application

One of the things that makes Qt so powerful is its property system. It is quite trivial to create properties for important parts of the application, and then save and restore them via the *QSettings* class. Consider utilizing *QSettings* for things like:

- Theme
- Language
- Current View
  - This is something that is quite important. Any application should have a structured way to access views so that a user can quickly access them without having to complete a series of steps.
- Any other stateful information that will be needed after the QML GUI layer is destroyed and recreated.
  - Avoid storing state in your QML files. QML is the GUI layer of the application and is not suited to maintain state.

Working with the UX team

With this functionality built into the application, it becomes more accessible to other members of the team, specifically UX. Adding the runtime reload functionality allows for:

- Bringing the design team into the development process to see and refine different design ideas by manipulating the QML layer.
  - Editing properties and design elements is very similar to working with CSS, which most design teams are familiar with.
- Working in real time on the target devices to evaluate the effectiveness of different design options quickly.

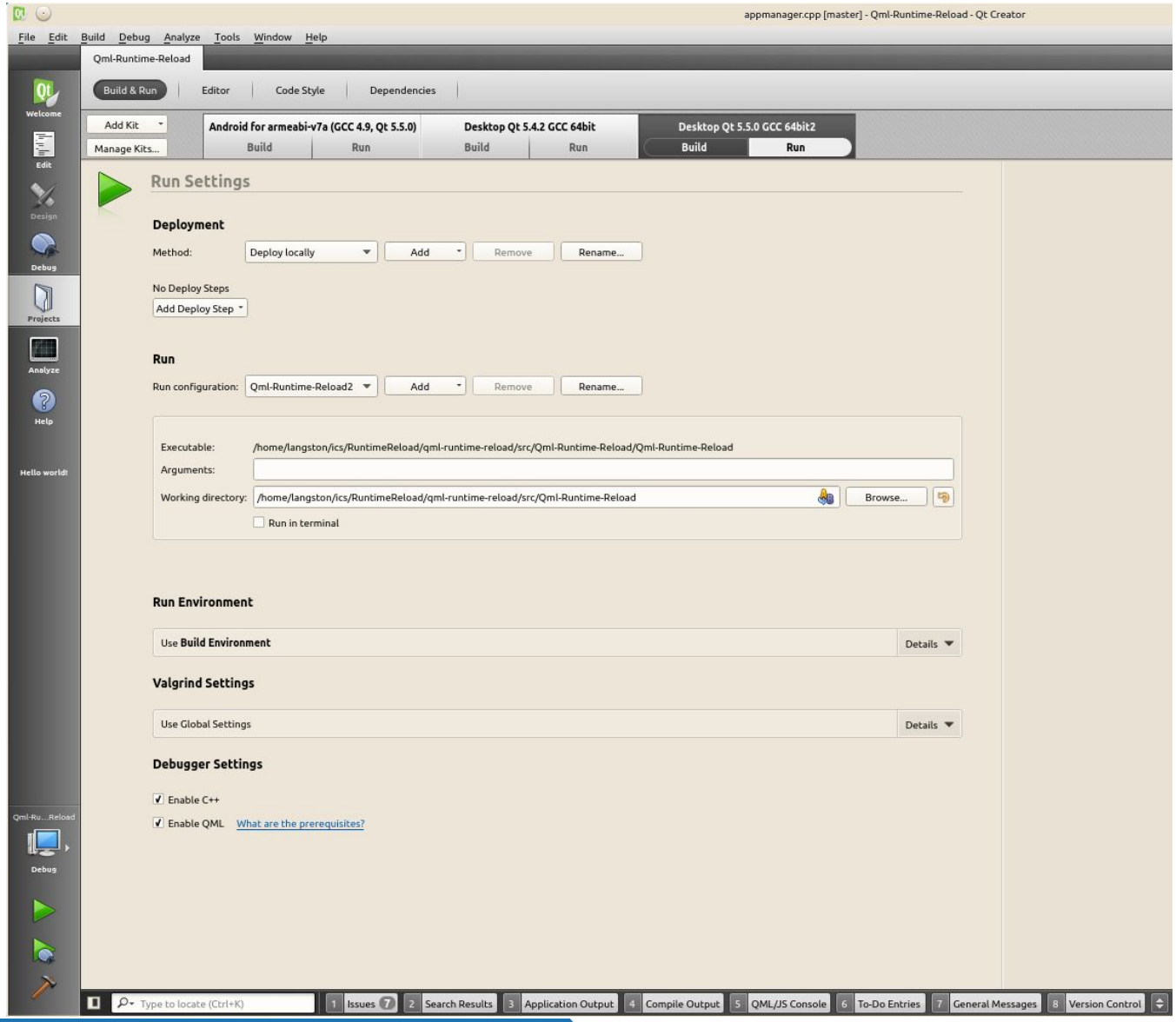
OS Considerations

Qt is an excellent multi-platform c++ toolkit. The wide array of platforms it runs on is a testament to this. And while the Qt contributors do their best to shield the developer for OS specific details, some things are inescapable.

File paths and reloading from disk:

- On Windows and OS X it can be helpful to set the ‘working directory’ in the run settings of QtCreator
- Linux tends to be the easiest to locate QML files within the source tree. You can provide a relative path to load the QML files.
  - If you want to use Shadow Building, make sure to escape the shadow build directory and enter the actual source directory
- On Windows, you will need to do some file path adjustments to find the correct source path.
  - in this case, it's just set to ‘qml’
- On OS X, a bundle is created. However, you don't want to edit the bundle, making this operation a little more difficult. You will need to obtain the path to your source tree
  - A common solution is to checkout the source to a designated location on the machine, so it will always be in the same place.

Run Settings



Summary

Reloading your QML GUI layer at runtime is a useful feature for different aspects of development and production. By implementing this functionality, you can make the application more flexible in terms of what you can do with your UI, to shaving time off of building the UX from a spec to being able to provide a robust set of themes for the end user.

Remember:

- Store important state information in c++, not QML
- Use *QSettings* (or your own persistent storage mechanism) to persist this state between sessions of the application
- Utilize *QFileSelector* / *QQmlFileSelector* where possible (if theming or styling)
- Don't couple your GUI to the c++ backend
    - This requires extra overhead to cleanup if you ever decide to change the look of the QML GUI. Use properties and *Q_INVOKABLE* functions to create an API and separation between the QML GUI and the backend.

Fin

Thank You

Anyone have questions?