Qt Quick – Hybrid models and Mobility

Timo Strömmer, Jan 10, 2010

Contents

<symbio>

Simple C++ / QML integration example

C++/QML HYBRIDS

Hybrid programs

<symbio>

- Hybrid programs get the benefit from both worlds
  - Ease of QML / script programming
    - See for example *hellographics* vs. *HelloGraphicsQML*
  - Power and flexibility of C++
    - Access to all services provided by the platform
    - C++ performance with larger data-sets

QML/C++ hybrid

<symbio>

- A C++ GUI application may contain *QDeclarativeView* GUI components
  - Available for example via the GUI designer
  - Each runs it's own declarative engine
    - *qmlviewer* also runs one
  - Resource-wise it's not a good idea to run many views in a single program

QML/C++ hybrid

- *QDeclarativeView* has *setSource* function, which takes the URL of the QML file as parameter
  - Thus, can load also from web
- The QML files of the application can also be bundled into a resource file

QML/C++ exercise

<symbio>

- Create a new Qt GUI Project
  - Add a *QDeclarativeView* to the GUI form
  - Add *QT += declarative* to .pro file
    - Takes declarative QT module into use
  - Add a QML file to the project
    - Implement a GUI
  - Add a new *Qt resource file* to project
    - Add a / prefix
    - Add the QML file under the prefix

QML/C++ exercise

<symbio>

- Load the QML file from resource in the MainWindow constructor
- Build and run

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    ui->declarativeView->setSource(QUrl("qrc:/Main.qml"));
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

QML/C++ interaction

<symbio>

- To access the QML core from C++ side, the *QDeclarativeView* exposes a *root context*
  - *QDeclarativeContext* class
- A property can be set with *setContextProperty* function
  - Access normally by name in QML

```
Rectangle {
    width: 300
    height: 200

    Rectangle {
        x: 25; y: 25; widt
        color: rectColor
```

```
QDeclarativeContext *context = ui->declarativeView->rootContext();
context->setContextProperty("rectColor", QColor(Qt::blue));
```

rectColor becomes property of root QML element

Exporting objects to QML

<symbio>

- Objects are registered with *qmlRegisterType* template function
  - Object class as template parameter
  - Function parameters:
    - *Module name*
    - Object version number (major, minor)
    - Name that is registered to QML runtime

Details about modules from:
http://doc.trolltech.com/4.7/qdeclarativemodules.html

```
qmlRegisterType<LocationNotifier>("LocationExample", 1, 0, "LocationNotifier");
```

Using exported classes

- The exported classes can be used as any other QML component
  - The module needs to be imported

```
qmlRegisterType<LocationNotifier>("LocationExample", 1, 0, "LocationNotifier");
```

```
class LocationNotifier : public QObject
{
    Q_OBJECT

    Q_PROPERTY(QString text READ text WRITE setText NOTIFY textChanged)
```

```
import Qt 4.7
import LocationExample 1.0

Rectangle {
    width: 200
    height: 200

    LocationNotifier {
        id: notifier
    }

    Text {
        x: 66; y: 93
        text: notifier.text
    }
}
```

QML object visibility

<symbio>

- Visibility at QML side
  - QObject *properties* become element properties
    - *on<Property>Changed* hook works if the NOTIFY signal is specified at C++ side
      - Also note that C++ signal name doesn't matter
  - QObject *signals* can be hooked with *on<Signal>*
  - QObject *slots* can be called as JS functions

<symbio>

QML plug-in projects

HYBRID PROGRAMMING

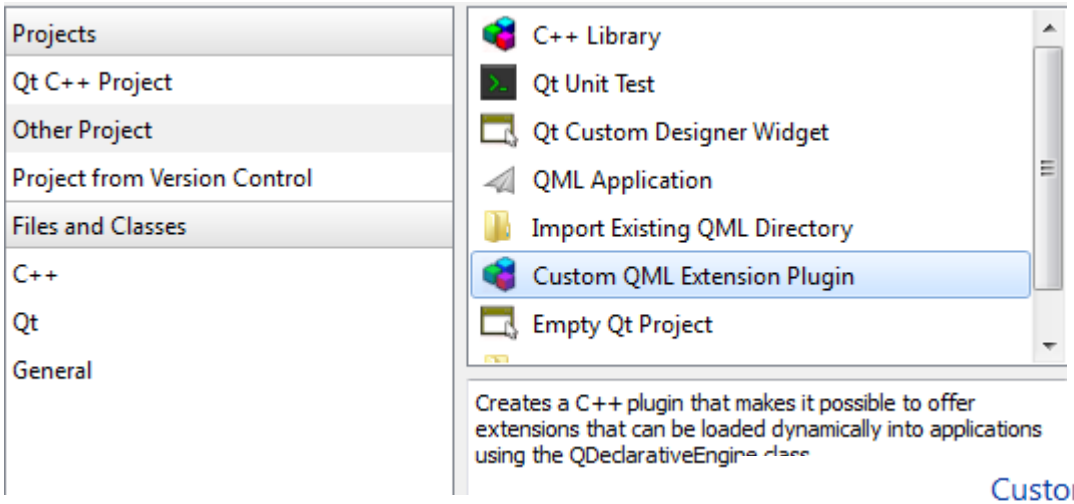
QML plug-ins

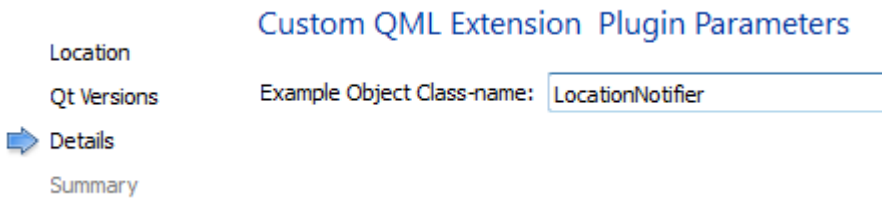
<symbio>

- A plug-in allows QML runtime to load Qt/C++ libraries
  - Thus, QML/C++ hybrid code can be run via *qmlviewer* or some other QML launcher application

Quick start

- Create a QML extension plug-in project
  - Wizard generates one *QObject*-based class





QML plug-in basics

- A QML plug-in library must have a class which extends *QDeclarativeExtensionPlugin*
  - Wizard generates

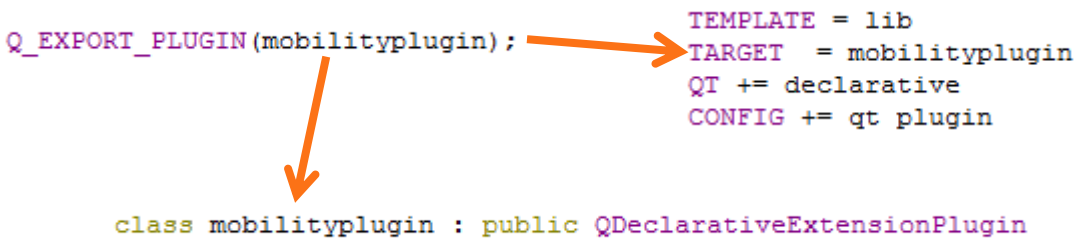
```
class mobilityplugin : public QDeclarativeExtensionPlugin
{
    Q_OBJECT

public:
    void registerTypes(const char *uri);
};
```

  - The plugin has *registerTypes* function, which is used to define components that are exported to the QML runtime
    - Use *qmlRegisterType* and pass the *uri* to the module parameter

QML plug-in basics

- The API used by QML runtime to load the plug-in is created via preprocessor *macro*
  - *Q_EXPORT_PLUGIN* if plug-in project and class names are the same (wizard does that)
  - *Q_EXPORT_PLUGIN2* if names are different
    - *See qmlpluginexample* directory



QML plug-in basics

- The plug-in must define a *qmldir* file
  - Describes the name of the plug-in to load
    - *libmobilityplugin.so* on Linux
    - *mobilityplugin.dll* on Windows
    - Optionally may specify a sub-directory

qmldir

```
1 plugin mobilityplugin
2
```

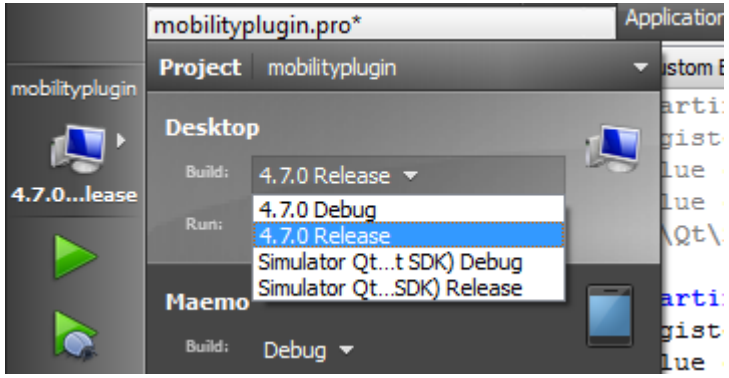
Quick start continued

- Create a QML application project
  - Copy the *qmldir* file from the C++ plug-in into the applications *plugin* directory
- Edit the C++ plug-in *.pro* file
  - Add *DESTDIR* statement to point to the QML application directory

```
DESTDIR = ../MobilityExample/plugin
```

Quick start continued

- Switch to *Release* build
  - Fails with *Debug* libraries



- Build
  - The *.so* or *.dll* should be in the QML application directory where *qmldir* file also sits

Example plug-in

<symbio>

- See *PluginExample* and *qmlpluginexample* directories
  - Stores *TextInput* content into a file while user is typing
  - Uses *QSettings* API from Qt core
    - File in ~/.config/Symbio/QmlPluginExample.conf

More QML plug-in functions

HYBRID PROGRAMMING

Adding object properties

<symbio>

- Root QML context is also available to plug-in components
  - The *QDeclarativeExtensionPlugin* may implement *initializeEngine* function
  - *QDeclarativeEngine* which runs the QML code comes as function parameter

```
void qmlpluginhooks::initializeEngine(QDeclarativeEngine *engine, const char *uri)
{
    QDeclarativeContext *context = engine->rootContext();
    context->setContextProperty("cppProperties", new RootProperties(this));
}
```

Running pure QML

<symbio>

- Sometimes it’s better to test in pure QML environment within *qmlviewer*
  - Speeds up GUI development
- Two ways
  - Use *dummy objects* to replace C++ types
  - Use *dummydata* directory to replace properties assigned from C++

Dummy types

- To create a dummy type, add a corresponding QML component into the *plugin* subdirectory
  - Loaded if *qmldir* is missing

```
ExampleObject {
    id: exampleObj
}

Text {
    x: 66
    y: 93
    text: exampleObj.text
}
```

```
QtObject {
    property string text: "Dummy"
}
```

```
class ExampleObject : public QObject
{
    Q_OBJECT

    Q_PROPERTY(QString text READ text WRITE setText NOTIFY textChanged)
```

Dummy properties

- *qmlviewer* supports loading of properties from *dummydata* subdirectory
  - The file name must equal the *property name* and thus starts with *lower-case letter*
  - Properties must be wrapped into *QObject*

```
QDeclarativeContext *context = engine->rootContext();
context->setContextProperty("cppProperties", new RootProperties(this));
```

```
Rectangle {
    width: 200
    height: 200

    color: cppProperties.color
```

cppProperties.qml | QtObject

```
import Qt 4.7

QtObject {
    property color color: "lightGray"
}
```

Plug-in example

<symbio>

- Example in *qmlpluginhooks* and *PluginHooks* directories
  - Registers a QML type
  - Registers properties into *root context*
  - Uses dummy data for both

Setting up mobile development environment

MOBILE DEVELOPMENT

Mobile development

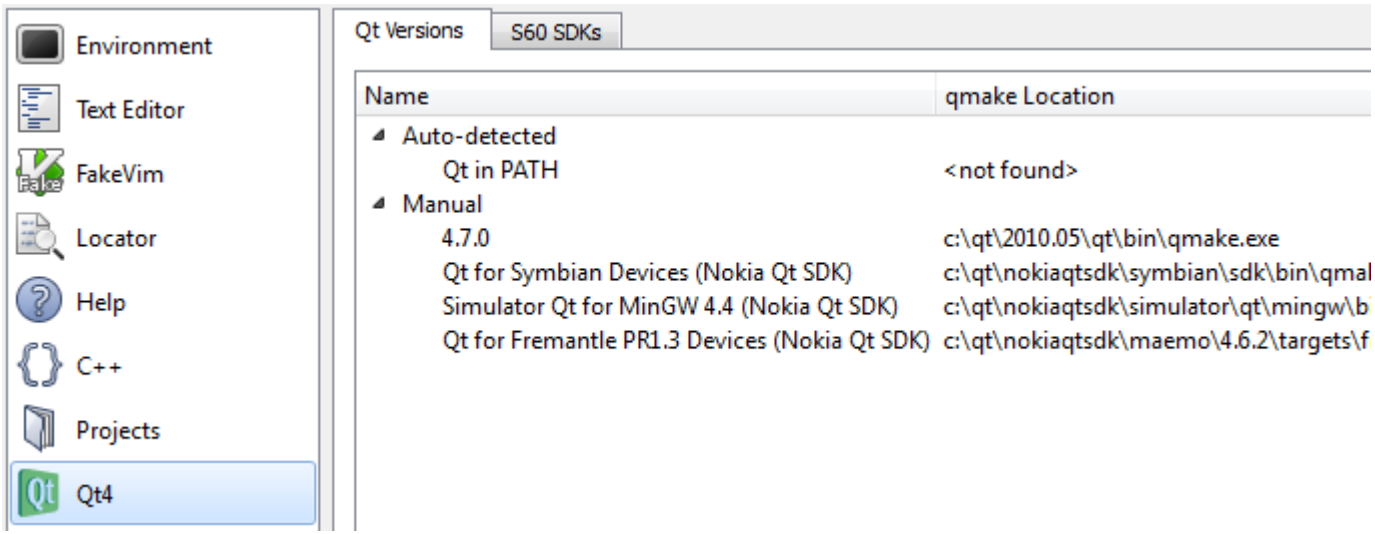
<symbio>

- Mobile devices run different instruction set
  - C++ code needs to be *cross-compiled*
  - Qt SDK can build for N900 and Symbian
- Debugging on mobile difficult
  - A simulator environment provided by Nokia
- Focus on N900 in these slides

Mobile targets

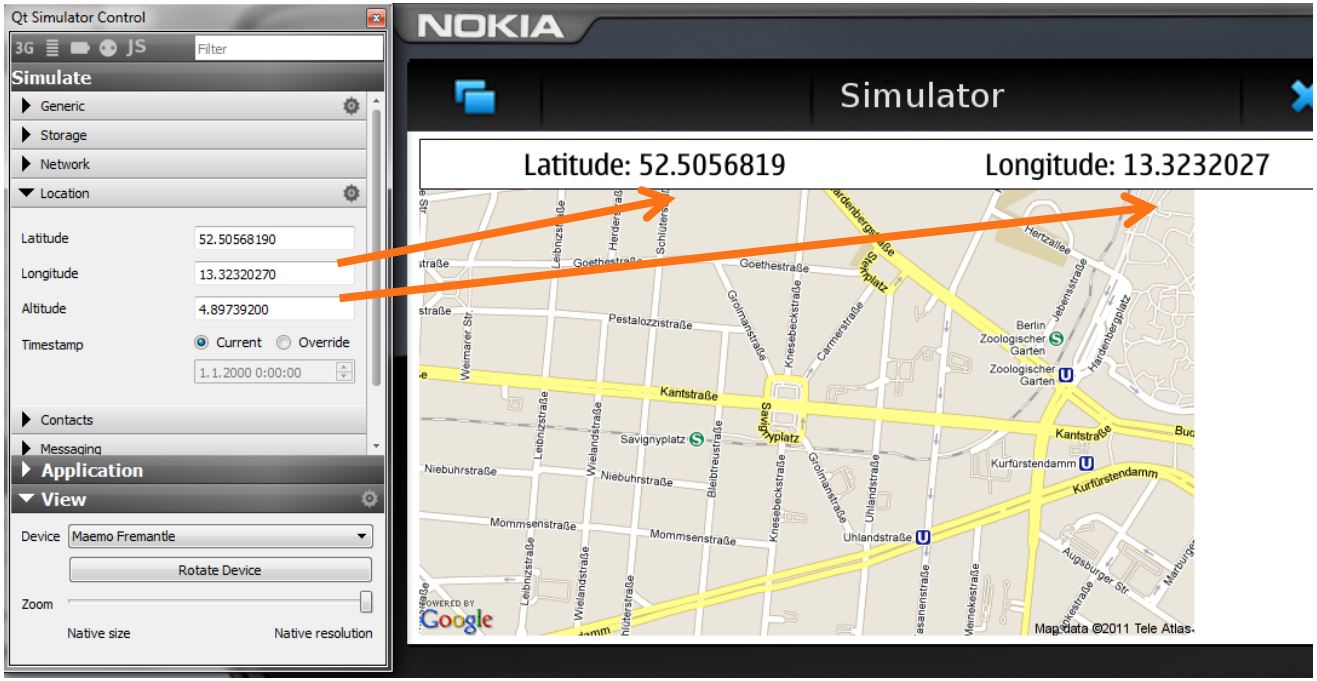
<symbio>

- Three pre-installed targets in SDK
  - Maemo, Symbian, Simulator



Qt Simulator

- Simulator target can be used to test N900 or Symbian projects without real device



N900 environment

- Install *Nokia Qt SDK*
  - Or integrate MADDE to existing QtCreator
- Install required libraries on the device
- Setup usb network between devices
- Setup SSH connectivity
- Build and run

Environment setup

<symbio>

- N900 guide at:
  - http://doc.qt.nokia.com/qtcreator-2.0.1/creator-developing-maemo.html
- Additionally QML viewer and Qt Mobility
  - http://paazio.nanbudo.fi/tutorials/qt-quick/qt-quick-qml-viewer-installation-in-nokia-n900

N900 packaging

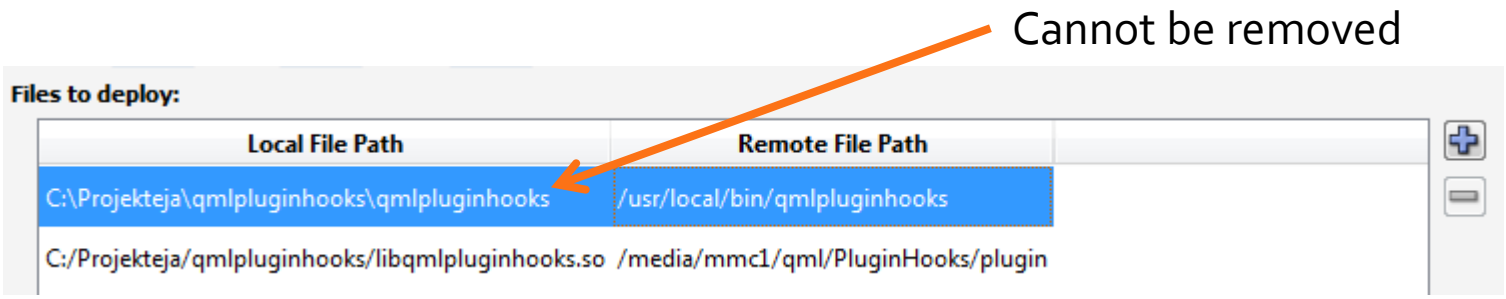
- QtCreator project options have an additional *packaging* step for Maemo projects

N900 packaging

- QtCreator can only package applications
  - By default the project goes to */usr/local/bin*
  - QML must be packaged into project resources
- Pure QML projects must be deployed by different means
  - For example memory card copy
  - See also http://qml.haltu.fi/

N900 packaging

- For plug-in projects, just disable packaging
  - You'll still get the library *.so*
    - Copy to device into project directory
  - Note that you cannot run from memory card



N900 packaging notes

- Mainly for own internal testing
  - No applications menu integration etc.
    - N900 based on *Hildon* UI framework
  - Better wait for MeeGo and next Qt SDK
- Also note that MeeGo uses different packaging mechanism
  - Debian in Maemo, RPM in MeeGo

Overview of the mobility API's

QT MOBILITY

Qt Mobility API's

<symbio>

- List of API's in Qt Mobility 1.1
  - http://qt.nokia.com/products/qt-addons/mobility
- Pre-built libraries in SDK
  - But not the latest 1.1 version
    - Missing some libraries and QML integration
  - Can be tested in *Simulator*
  - To get latest, you'll need to build from sources

Qt Mobility API's

<symbio>

- Mobility API's are enabled via project file
  - *CONFIG += mobility*
  - *MOBILITY = <modules to use>*
- Example in *n900test* directory
  - For some reason the *qmllocation* project refuses to work

Domain	Value
Bearer Management	bearer
Contacts	contacts
Location	location
Multimedia	multimedia
Messaging	messaging
Publish And Subscribe	publishsubscribe
Service Framework	serviceframework
Sensors	sensors
System Information	systeminfo
Versit	versit
Document Gallery	gallery
Organizer	organizer
Tactile Feedback	feedback

```
TEMPLATE = app
CONFIG += mobility
MOBILITY = location
```

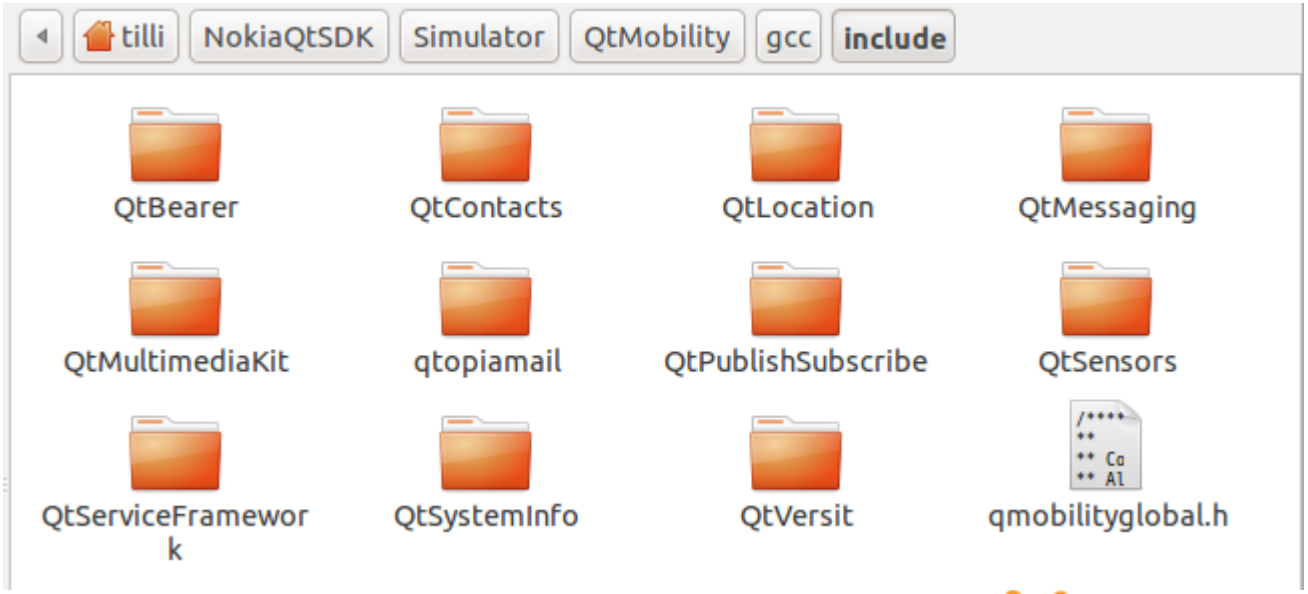
Qt Mobility API's

	S60 5th Edition	Symbian	Maemo 5	Harmattan	Windows XP/Vista	Linux	Mac OS X
Service Framework (in-process)							
Messaging							
Bearer Management							
Publish and Subscribe							
Contacts							
Location							
Multimedia							
System Information							
Sensors							
Versit(vCard)							
Versit(Organizer)							
Camera							
Service Framework(OOP)							
Organizer							
Landmarks							
Document Gallery	*)						
Maps/Navigation							
Feedback							

Qt Mobility API's

<symbio>

- SDK comes with older libraries
  - So, check what's in there
  - *<creator-path>/Simulator/QtMobility/gcc/include*



<symbio>

QML / Web hybrids

HYBRID PROGRAMMING

Web browser integration

<symbio>

- QML has *WebView* component
  - Was used in *n900test* example to load map
  - Found from *QtWebKit 1.0* module
- Basic properties
  - *url* specifies the web page to display
  - *preferredWidth* and *preferredHeight* to control the displayed web page size
  - Wrap into *Flickable* to enable scrolling
    - Still no scrollbars…

Web browser integration

- *WebView* provides *action* objects for web page navigation
  - *back*, *forward*, *reload* and *stop*
  - Actions can be *triggered* from script

```
DemoButton {
    text: "F"
    onClicked: webView.forward.trigger();
}
```

- Basic hook signals
  - *onLoadStarted*, *onLoadFinished*, *onLoadFailed*
  - *onAlert*

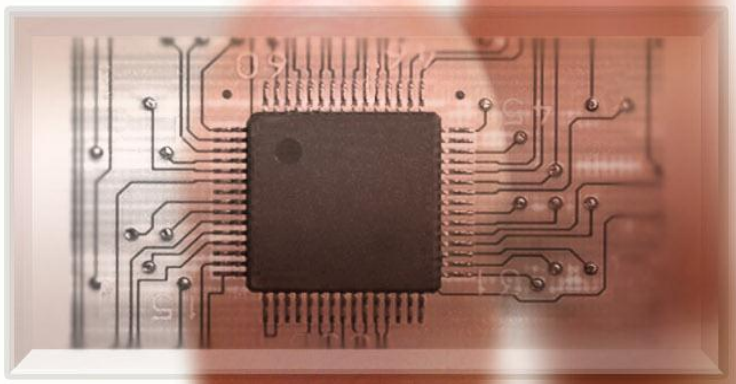
Web browser integration

<symbio>

- QML and web scripts may interact with each other
  - *evaluateJavaScript* function runs script code in the context of the web browser
  - *javascriptWindowObjects* property registers QML objects as properties of the web *window*
    - *WebView.windowObjectName* attached property defines the name visible to browser
    - Functions of the object can be called via browser script
- Example in *WebHybrid* directory

<symbio>

QUESTIONS?



SERIOUS ABOUT SOFTWARE