Qt IVI: Integrating & Testing vehicle functions with Qt Automotive Suite

Mike Krus, Senior Software Engineer at KDAB
QtIVI
- What is Qt Auto
- QtIVI - Extendable Cross Platform APIs
- Code Generation
- Integration with Gammaray

What is Qt Auto - 1
- Qt HMIs
  - easy to implement stunning user interface with seamless integration of 2D and 3D content
  - scales to different hardware and can leverage GPU-based rendering for smooth 60fps experience even on high-resolution screens
  - easy to use declarative UI description language with graphical tooling, as well as C++ APIs for full native power where needed

- Qt Automotive Suite
  - modular vehicle data and multimedia API with comprehensive simulation backends enable development before hardware is available
  - multi-process architecture with app lifecycle management and security enabling a modern modular HMI design and safe integration of 3rd-party applications
  - reference HMI implementation

What is Qt Auto - 2
- Tooling
  - support for emulated or on-target development
  - support for quick target deployment and live development and debugging on target
  - high-level and visual diagnostic tools enabling effective analysis of complex bugs or performance issues

- Custom SDK
  - ensure everyone using exactly the same setup to avoid integration problems
  - single installer with online update support to efficiently deploy your development setup
  - include in-house or 3rd party frameworks
What is QtiVI

- Feature Abstraction
  - Provide support for multiple features: climate control, media services, ...
  - Consistant frontend API
  - Multiple manufacturer backend
- Core Library
  - Abstract features, including support for zoning
  - Frontend / Backend setup with dynamic plugin loading
  - Target multiple configurations for deployment, simulation, testing, ...
- C++ and QML interface
- Reference implementations

What is QtiVI - Architecture

Features and Services

- A Feature is an API to a subset of functionality
  - derived from QIviAbstractFeature and QIviAbstractZonedFeature
- Backed by a Service which implements the required interface
  - derived from QIviFeatureInterface and QIviZonedFeatureInterface
- Features are have different backends which are discovered at runtime
  - Call QIviAbstractFeature::startAutoDiscovery()
  - Possible to set the discovery mode using QIviAbstractFeature::setDiscoveryMode(DiscoveryMode discoveryMode)
  - Supports AutoDiscovery, LoadOnlyProductionBackends, LoadOnlySimulationBackends
  - Or use QIviServiceManager to find required service object and the right feature

Features have properties, signals, slots
- Data and functionality is implemented in backend service
- Backend service may perform validation on incoming values
- Will notify front end for state changes

```cpp
1  int QIviClimateControl::fanSpeedLevel() const
2  {
3    Q_D(QIviClimateControl);
4    return d->m_fanSpeedLevel;
5  }
6
7  void QIviClimateControl::setFanSpeedLevel(int fanSpeedLevel)
8  {
9    if (QIviClimateControlBackendInterface *backend =
10       qobject_cast<QIviClimateControlBackendInterface*>(this->backend()))
11      backend->setFanSpeedLevel(fanSpeedLevel, zone());
12  }
```
Features and Services

- Features have properties, signals, slots
- Data and functionality is implemented in backend service
- **Backend service may perform validation on incoming values**
- Will notify front end for state changes

```cpp
1 void QIviClimateControlBackend::setFanSpeedLevel(int fanSpeedLevel)
2 {
3     if (m_fanSpeedLevel == fanSpeedLevel)
4         return;
5     if (fanSpeedLevel <= 50)
6         m_fanSpeedLevel = fanSpeedLevel;
7     emit fanSpeedLevelChanged(m_fanSpeedLevel);
8 }
```

Zoned Features

- Features and their properties can be zoned
- Used for features that are available in multiple zones of the vehicle
- List of zones are accessible via QIviAbstractZonedFeature::availableZones()
- Zone objects are accessible via QIviAbstractZonedFeature::zoneAt()

```cpp
1 QIviClimateControl* climateControl = new QIviClimateControl(QString(), this);
2 climateControl->startAutoDiscovery();
3 QIviClimateControl* frontLeftControl = climateControl->zoneAt("FrontLeft");
```

- All zone objects talk to the same backend service instances
- Some properties may be zoned, some not

QtiVI 1.X: QiviProperties

- Properties are stored as QIviProperty objects, values are stored in QVariant
- Use QIviProperty::value() to get the current value, QIviProperty::setValue() to set it
- Possible to define minimum and maximum values via virtual functions QIviProperty::minimumValue() and QIviProperty::maximumValue()
- QIviProperty::availableValues() return the accepted values
- QIviProperty::isAvailable() gives information about the property being available in the backend
- Property attributes are accessible as QML grouped properties, like myProperty.available, myProperty.value, etc.

**Deprecated**

- Not type safe
- Lots of overhead
- Valid ranges only apply to small subset of properties
- Availability tends to be static
Writing your own

- Lots of code to write
  - Front end, all the properties, listening to backend changes
  - One or more backends
  - Production backend talking to the actual vehicle
  - Simulation backend handling mock data
  - Handling per zone values, valid ranges, etc
- Repeat for each interface

Could this be automated?

IVI Generator

- QFace - https://github.com/Pelagicore/qface
- IDL designed to support
  - Qt related features such as properties, signals, slots, models...
  - basic data types, structs, enums and flags
  - annotation to provide meta-data about modules, interfaces, properties...
  - structured comments
- Generator
  - Python3 + ANTLR based parser
  - Jinja2 based templates
  - Walk the domain model and produce the required output using the templates
- Templates
  - Frontend and backend templates to generate code

QtIVI 2.0: Code Generation

- Define interface
  - Name, properties, signals, slots
  - Associated enums and structures
  - Meta data about ranges, valid values, default values, zones...
- Use generators
  - Frontend generator for the feature API and abstract backend interface
  - Backend simulator generator supporting default values, validity checks, etc

QFace Sample

```python
1 module org.example 1.0
2
3 interface Echo {
4     string message;
5     readonly Status status;
6     void echo(string message);
7     signal broadcast(string message);
8 }
9
10 enum Status {
11     Null, Loading, Ready, Error
12 }

13 class Echo : public QObject
14 {
15     Q_OBJECT
16     Q_PROPERTY(QString message READ message WRITE setMessage NOTIFY messageChanged)
17     Q_PROPERTY(Example::Status status READ status NOTIFY messageChanged)
18     public:
19         QString message() const;
20         void setMessage(const QString& message);
21         ...
22     Q_INVOKABLE void echo(QString message);
23     signals:
24         void messageChanged(const QString& message);
25         void broadcast(const QString& message);
26     }
27 ```
QFace Grammar

- `module <module> <version>`
- `import <module> <version>`
- `interface <Identifier> {` (with optional attributes)
  - `[const]` [readonly]` <type> <identifier>` (with optional parameters)
  - `signal <signal>(<parameter>*)` (with optional const attribute)
- `struct <Identifier>`
  - `<type> <identifier>;`
- `enum <Identifier>`
  - `<name> [= <value>]`
- `flag <Identifier>`
  - `<name> [= <value>]`
- `Builtin types: bool, int, real, string, var`
- `Models (and lists)`
- `Structured (JavaDoc) comments`

Annotations

- All elements can be annotated
- `@singleton: true`
- `@config: { port: 1234 }` (with optional interface)

- Single value or compound
- Compound values use YAML syntax
- External annotations (.yaml file)

- `Only relevant to the generator`

QtIvi Example

- Entire module defined in single QFace file
- Defines common annotations

```
@config: { qml_name: 'QtIvi.VehicleFunctions', \n    interfaceBuilder: 'vehicleFunctionsInterfaceBuilder' }
module QtIviVehicleFunctions 1.0;
```

Generators

- Traverse the document model to generate files
- YAML file to describe which template file to use for each object in the document model

```
for module in system.modules:
    # generate module related files...
    for interfaces in module.interfaces:
        # generate interface related files...
    for struct in module.structs:
        # generate interface related files...

JINJA based template documents

- `frontend` creates QIviAbstractFeature and QIviAbstractZonedFeature based classes and helper
- `backend_simulator` creates QIviFeatureInterface and QIviZonedFeatureInterface based classes with all data and operations
- `control_panel` creates test app (more later)
Validation

- Annotation define valid ranges, minimum/maximum values, domains...

```cpp
interface QIviClimateControl {
    @config_simulator: { range: [0, 50] }
    int fanSpeedLevel;
    @config_simulator: { minimum: 0 }
    int steeringWheelHeater;
    @config_simulator: { maximum: 30.0 }
    real targetTemperature;
    @config_simulator: { domain: ['cold', 'mild', 'warm'] }
    string outsideTemperatureLabel;
}
```

- Used in backend_simulator generator to test incoming values

- Information is stored in class meta data as JSON

```cpp
class QIviClimateControl : public QIviAbstractZonedFeature {
    Q_OBJECT
    Q_CLASSINFO("IviPropertyDomains", "{
        "iviVersion":2.0, "fanSpeedLevel": {"range":[0,50]}
        "steeringWheelHeater": {"minimum":0}
        "targetTemperature": {"maximum":30.0}
        "outsideTemperatureLabel": {"domain": ["cold","mild","warm"]}
    }")
public:
    ...
}
```

Customizing Backend

- Backend classes only contain default data members
- Operations only have default implementations
- Full functionality needs specialisation
- Can provide `generator` function to create derived instances
- Annotation applied to module:
  @config: {interfaceBuilder: "vehicleFunctionsInterfaceBuilder"}

```cpp
extern QVector<QIviFeatureInterface *> vehicleFunctionsInterfaceBuilder(QIviVehicleFunctionsPlugin *plugin)
{
    const QStringList interfaces = plugin->interfaces();
    QVector<QIviFeatureInterface *> res;
    Q_ASSERT(interfaces.size() == 2);
    Q_ASSERT(interfaces.indexOf(QIviVehicleFunctions_QIviClimateControl_iid) == 0);
    Q_ASSERT(interfaces.indexOf(QIviVehicleFunctions_QIviWindowControl_iid) == 1);
    res << new QIviClimateControlBackend(plugin);
    res << new QIviConcreteWindowControlBackend(plugin);
    return res;
}
```

QMake integration

- QMake has been extended to support QFace files and drive the code generation
- Generator creates code and .pri files, not overall project

```cpp
CONFIG += ivigenerator
QFACE_FORMAT = backend_simulator
QFACE_SOURCES = qivivehiclefunctions.qface
QFACE_MODULE_NAME = QtIviVehicleFunctions
```

Customizing Templates

- Can more code be generated?
  - For backend, can more low level integration code be generated?
  - Add special annotations and customize the templates!
- QFACE SOURCES can be a path to a template folder, requires a matching YAML file
  - Generate tests, sample UIs, ...

```cpp
extern QVector<QIviFeatureInterface *> vehicleFunctionsInterfaceBuilder(QtIviVehicleFunctionsPlugin *plugin)
```
QtSimulator Integration

- Based on QtSimulator
  - Part of Boot2Qt (not the old simulator-qt, not the new qt-emulator)
  - Used by various Qt modules (location, ...)
- control panel template generates application
  - Mirrors backend behaviour
  - Useful for testing application in absence of full featured backend
Overriding Properties

- Gammaray's IVI module provides introspection
- Observe and modify property values from QIVIAbstractFeature derived classes
- Also support **overriding**
  - Change values sent from the backend
  - Modify values without affecting the backend
- Useful to test application behaviour in isolation of any backend

Conclusion

- QtIVI, library and tools for integrating vehicle functions in Qt applications
- IDL and code generator massively simplify the creation of the glue code
- Tooling for testing, debugging and profiling

Thank you!

www.kdab.com

mike.krus@kdab.com