QtRobo Dokumentation

Programmiertechniken für Embedded Systems

Andreas M. Brunnet, B.Sc.

Inhaltsverzeichnis

Abstrakt

Bei der Entwicklung von Projekten auf Basis von Microcontrollern stellt das Testen der implementierten Funktionen meist einen größeren Aufwand dar. Hierzu werden Hardware-Komponenten zur Ein- und Ausgabe heran gezogen. Alternativ können entsprechende Funktionstests auch über eine serielle Schnittstelle vom Entwicklerrechner aus erfolgen. Auch in diesem Szenario wird oft auf einfache Software zurückgegriffen, mit der rohe ASCII oder Hexadezimale Daten gesendet und entsprechend als Text dargestellt werden können. Dem gegenüber stehen Programme wie RoboRemo [1], die es ermöglichen, auf einfache Art und Weise eine selbstdefinierte UI zu erzeugen, mit der entsprechend Embedded Devices getestet werden können.

Funktionalitäten

In diesem Abschnitt wird zum einen auf die Funktionalitäten sowie auch die Implementierungsdetails der Anwendung eingegangen. Beim Start von QtRobo findet man ein leeres Default Layout vor (siehe Abb. 1).

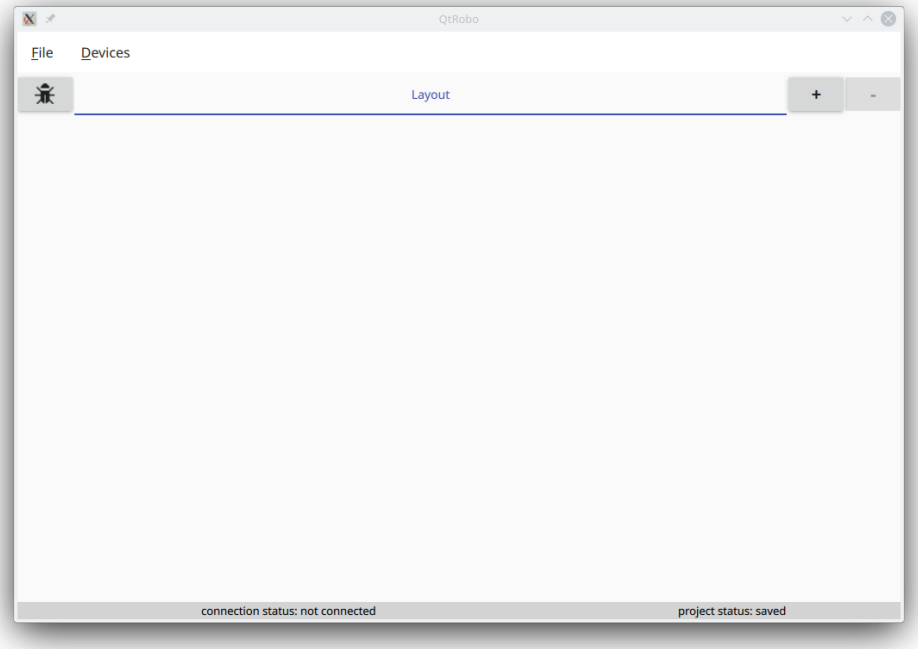

Abbildung 1. Default Layout

Mit einem Rechtsklick in das leere Layout öffnet sich das Kontextmenü (siehe Abb. 2). Über dieses Menü können die gewünschten Widgets ausgewählt und dem aktuell aktiven Layout hinzugefügt werden. Diese werden an der Stelle des Rechtsklick platziert. Widgets können nur im Editiermodus geändert werden. In diesem Modus können die Widgets nicht bedient werden. Die Positionierung der Widgets kann entweder mit oder ohne Einrastfunktion erfolgen.

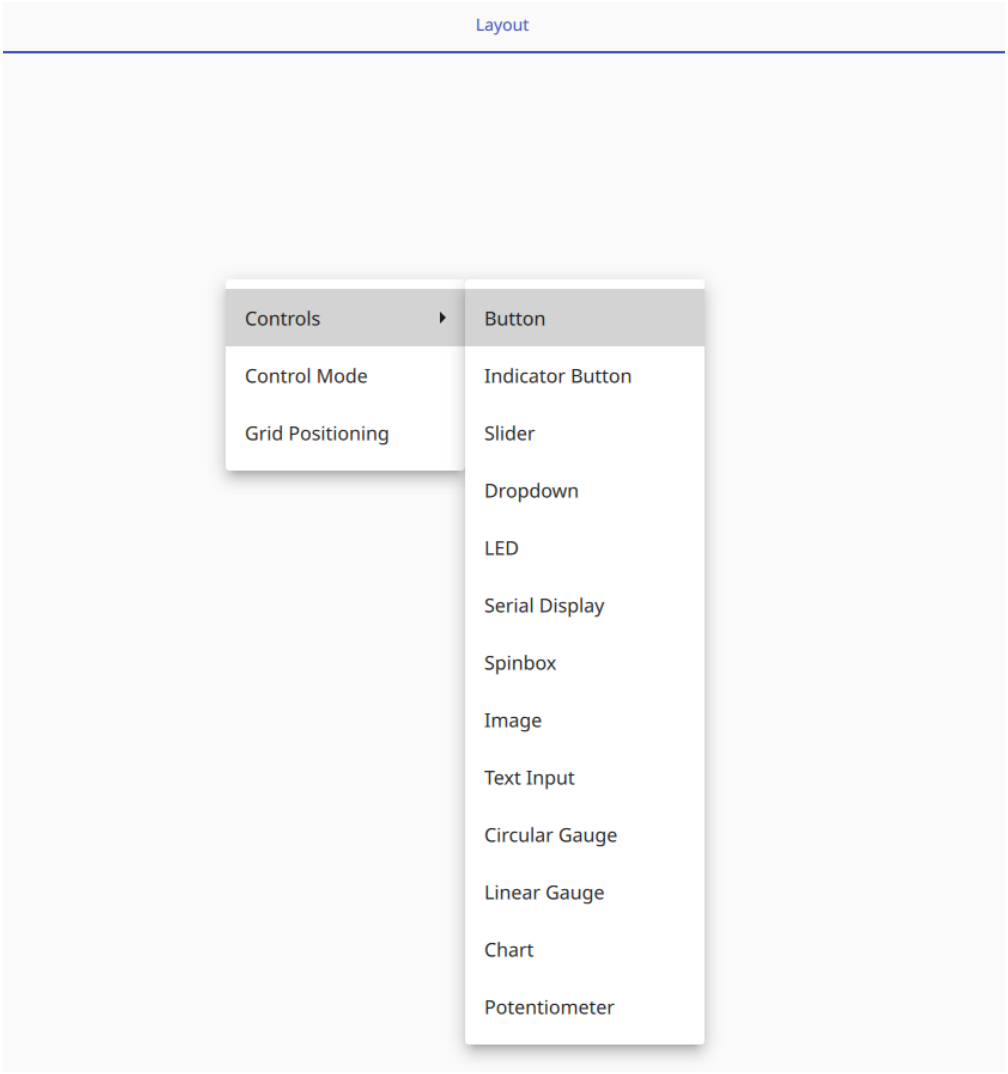

Abbildung 2. Kontext Menü

Nachdem ein Widget dem Layout hinzugefügt wurde, kann es über die Manipulatoren an den Ecken gelöscht, gedreht oder skaliert werden (siehe Abb. 3). Über Drag and Drop lässt sich ein Widget im Layout verschieben.

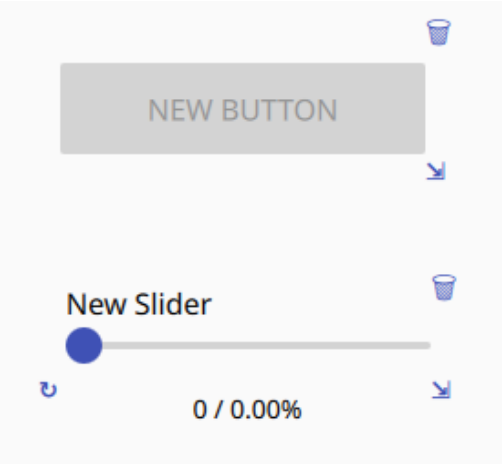

Abbildung 3. Widgets im Editiermodus

Die Eigenschaften eines Widgets können über dessen Kontextmenü geändert werden. Dies öffnet sich mit einem Rechtsklick auf das Widget (siehe Abb. 4).

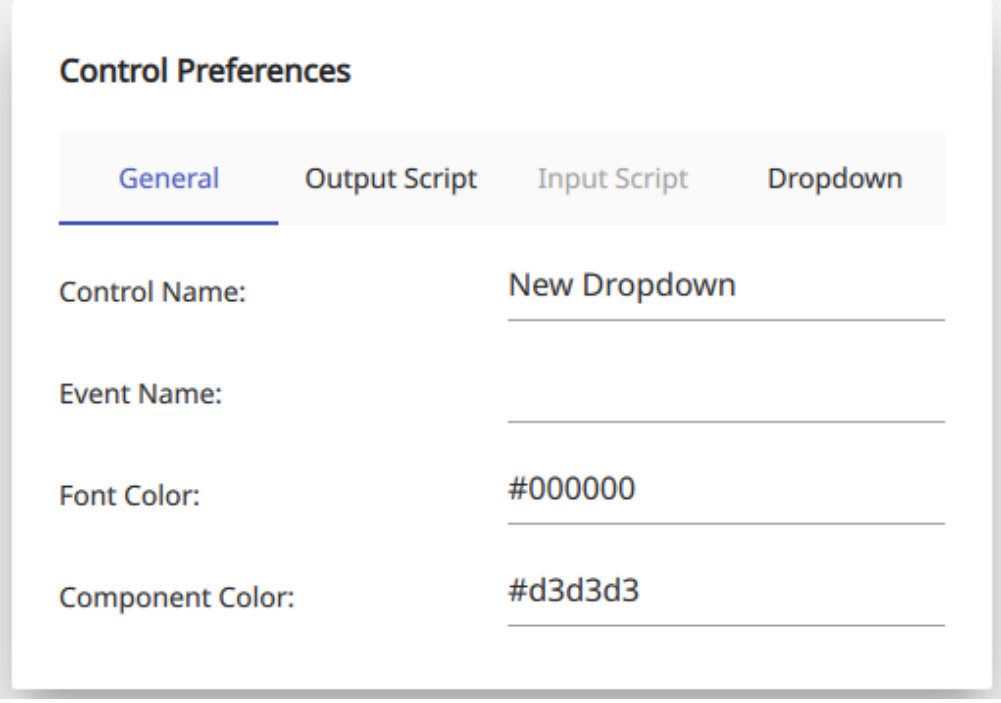

Abbildung 4. Widget Kontextmenü

Das Widget Kontextmenü besitzt vier Reiter. Der erste Reiter *Allgemein* definiert Eigenschaften, die jedes Widget besitzt. Der zweite Reiter definiert das Ausgabescript. Mit diesem lassen sich Wert und Eventname, die beim Aktivieren eines Widget gesendet werden über entsprechenden Javascript-Code manipulieren. Reiter drei funktioniert identisch zum zweiten. Jedoch können hier die eingehenden Werte, vor ihrer Darstellung durch das entsprechende Widget, manipuliert werden. Der letzte Reiter ist spezifisch für jedes Widget. Beispielsweise können dort für ein Dropdown-Widget neue Elemente zur Selektion angelegt werden.

Weitere Layouts können über die +-Schaltfläche, rechts neben dem Layout-Titel hinzugefügt werden (siehe Abb. Abb. 1). Mit der --Schaltfläche wird ein Layout gelöscht. Hierbei werden Layouts von rechts solange gelöscht, bis nur noch eines vorhanden ist. Der Name eines Layout kann über einen Rechtsklick auf den Layout-Titel geändert werden.

Links neben den Layout-Reitern befindet sich die Schaltfläche zum Öffnen des Debug-Fensters (siehe Abb. 1). Das Debug-Fenster zeigt eingehende und ausgehende Nachrichten an, bevor diese durch den Nachrichtenparser laufen (siehe Abb. 5).

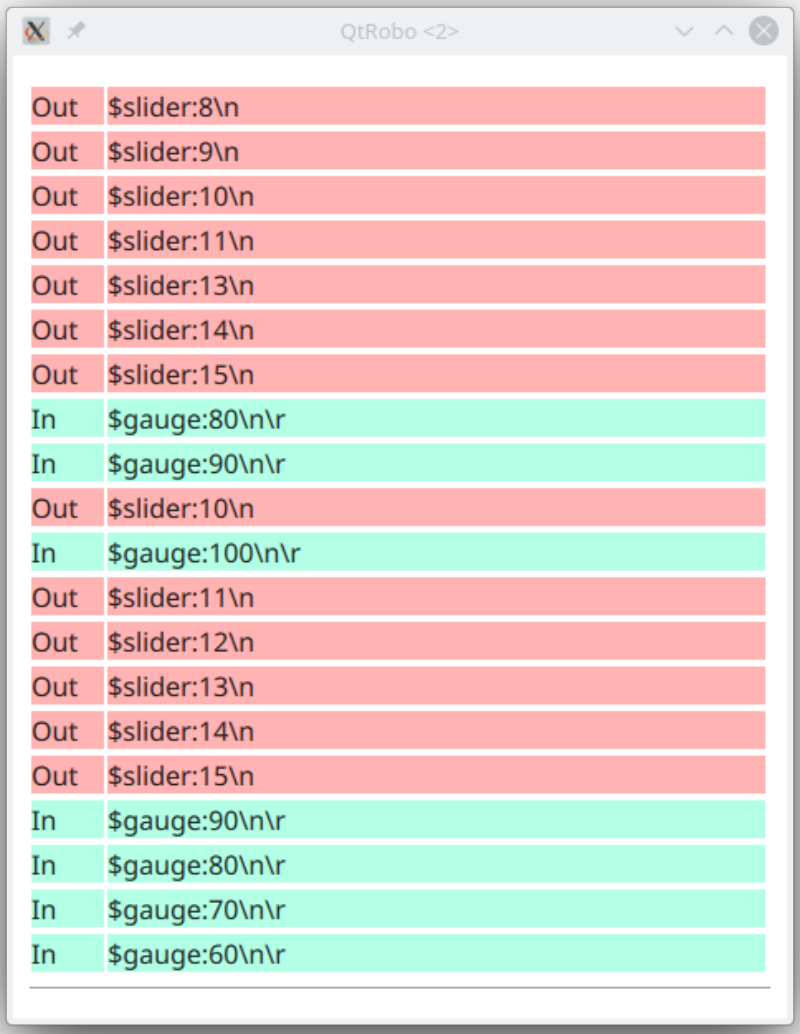

Abbildung 5. Debug Fenster

Über den Menüpunkt *Geräte* kann eine Verbindung entweder über einen seriellen Port oder über einen lokalen Socket hergestellt werden. Nach der Auswahl der Verbindungsart öffnet sich das Konfigurationsfenster der Verbindung (siehe Abb. 6).

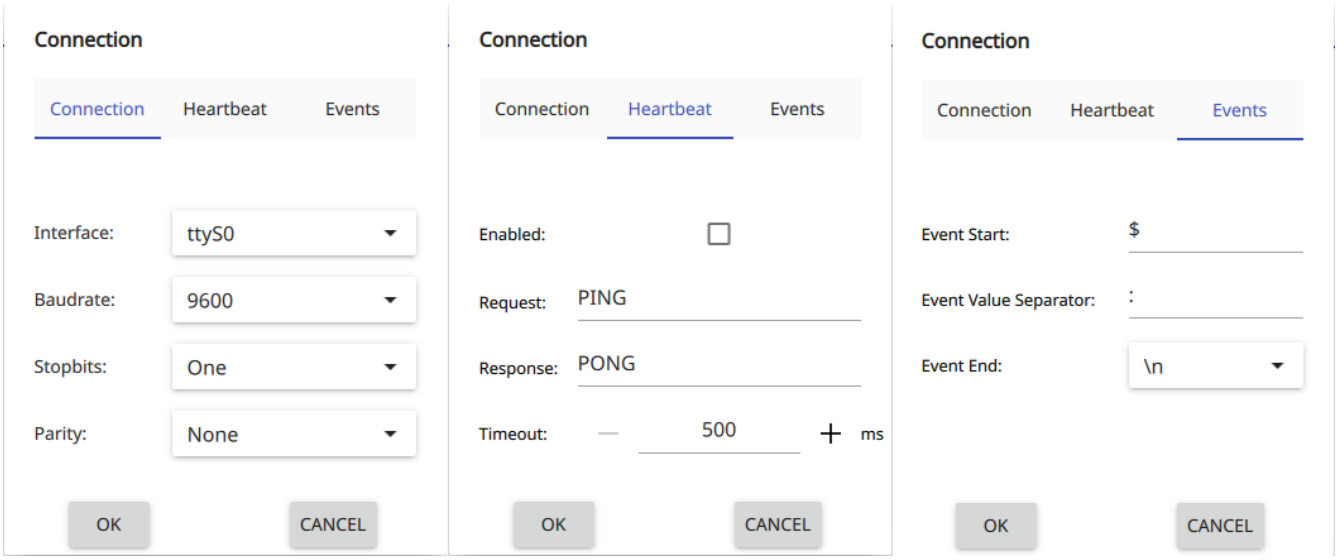

Abbildung 6. Verbindungskonfiguration

Zum einen können die verbindungsspezifischen Eigenschaften eingestellt werden. Des weiteren kann der Heartbeat konfiguriert werden. Dieser sendet periodisch eine Nachricht an den verbundenen Microcontroller / Middleware und zeig im unteren Teil des Hauptfensters an, wenn eine Antwort erscheint. Hierbei leuchtet eine Anzeige auf, die bei jeder erhaltenen Nachricht hellgrün aufleuchtet und allmählich in einen Rotton übergeht. Auch kann konfiguriert werden, wie die Nachrichten, die QtRobo sendet und empfängt, aufgebaut sind. Das bedeutet, welches Start-, Trenn- und Endesymbol genutzt werden soll. Über das *Geräte*-Menu kann die Verbindung auch wieder unterbrochen werden.

Der Menüpunkt *Datei* ist identisch zu anderen Applikationen. Es lässt sich ein neues Projekt anlegen, ein bestehendes öffnen, ein geöffnetes oder neues speichern.

Implementierung

QtRobo ist in C++ implementiert. In großem Umfang wird hierbei das QT Framework [2] eingesetzt. So werden, wenn möglich, die vom Framework angebotenen Klassen genutzt. Die UI und die entsprechende Logik zur Interaktion, wie dem Drag and Drop Mechanismus, sind in QML implementiert. QML bietet die Möglichkeit, im Vergleich zu anderen UI Frameworks, relativ einfach auch eigene UI Elemente zu definieren. Zusätzlich zum Qt Core werden noch folgende Abhängigkeiten verwendet.

- Serial Port
- Quick Controls 2
- Charts
- Scripts

Frontend

Das Frontend besteht aus QML-Modulen. Bei QML handelt es sich um eine deklarative Sprache zur Entwicklung von grafischen Oberflächen. Stark in QML integriert ist JavaScript. Entsprechend mächtig und flexibel können UI-Elemente erstellt werden. Jedoch verfügt QML über keine

Vererbung oder ähnliche Mechanik, was zu starker Codeduplikation führen kann. Dies findet sich auch in den implementierten Widgets (siehe Absch. Widgets) wieder.

Um auf Funktionen des Backends zuzugreifen, kann das globale Feld *qtRobo*, welches in der Main des Backends registriert wurde (siehe Bsp. Bsp. 4), genutzt werden (siehe Bsp. Bsp. 1).

Beispiel 1. Beispielhafter Zugriff auf Backend-Funktionen

```
qtRobo.connection.write()
qtRobo.connection.javascriptParser.runScript()
qtRobo.connection.heartbeatEnabled = false
```

main.qml

Als Einstiegspunkt der UI liefert das *main*-Modul das *ApplicationWindow* als Wurzel der UI. Neben dem Layout der Applikation, findet sich in der *main*, wie auch in allen anderen Modulen, ein hohes Maß an Darstellungslogik.

rootMouseArea

Das Drag and Drop von Widgets, aber auch der Aufruf des Kontextmenüs ist mit Hilfe einer *MouseArea* implementiert, die sich über die gesamte Größe des Layout-Bereichs des Projektes erstreckt. Zum Öffnen der Kontextmenüs wird lediglich das entsprechende Mausevent innerhalb dieser *MouseArea* abgefangen und geöffnet. QML besitzt bereits einen Drag and Drop Mechanismus. Dieser ist in jeder *MouseArea* verfügbar und stellt sich wie in Bsp. Bsp. 2 zu sehen dar.

Beispiel 2. QML Drag and Drop Mechanik

```
drag.target = myTarget
drag.axis = Drag.XAndYAxis
drag.minimumX = minX
drag.maximumX = maxX
drag.minimumY = minY
drag.maximumY = maxY
```

So wird zum Verschieben die gewünschte Komponente als *target* des *drag*-Objektes gesetzt. Danach kann über *axis* bestimmt werden, welche Axen betroffen sind. Das bedeutet, ob ein Objekt nur horizonal, nur vertikal oder in beide Richtungen verschoben werden kann. Der Bereich, indem das Objekt bewegt werden kann, wird mit den entsprechenden Minimum- und Maximum-Feldern für jeweils die X- und Y-Achse restriktiert. Damit ein Objekt wieder loggelassen wird, wird im *Release* -Event der Maus das *target* auf *undefined* gesetzt und das ursprünglich zugewiesene Objekt somit wieder freigelassen.

Die Funktionalität des Ausrichtens von Widgets anhand eines Rasters ist über das *PositionChanged* Event implementiert. Hierbei werden die X/Y-Koordinaten des Widgets, welche vom *drag*-Objekt implizit gesetzt werden, nachträglich explizit neu gesetzt. Es wird eine in der *GlobalDefinitions* (siehe Absch. GlobalDefinitions) gesetzte Konstante Modulo mit dem aktuellen Koordinatenwert genommen und das Resultat nochmals von diesem Koordinatenwert abgezogen. Dieser Endwert

entspricht dem schrittweisen Bewegen eines Widgets.

layoutToArray()

Zum Persistieren des Layouts eines Projektes bietet es sich durch die Integration von JavaScript in QML an, die Serialisierung der Layout-Informationen in QML selbst zu tätigen. Die Funktion liefert ein Array an Json-Objekten zurück. Dieses Array ist wie in Bsp. Bsp. 9 illustriert aufgebaut. Es wird über jedes Tab iteriert. Die Informationen über das Tab werden in einem Json-Objekt abgelegt. Danach wird über die Kinder des Tabs, also die Widgets, iteriert. Die Informationen der Widgets werden wiederum in einem Json-Objekt abgelegt und danach dem Arrayfeld *content* des Tab-Json-Objektes hinzugefügt. Am Ende wird das Tab-Objekt dem Tab-Array beigefügt. Nicht jedes Widget besitzt alle aufgelisteten Felder. Jedoch ist dies bei der Serialisierung irrelevant, da nicht vorhandene Felder den Wert *undefined* haben und einem Json-Objekt nicht hinzugefügt werden.

arrayToLayout()

Zum Laden des Layouts wird ein Tab-Array vom Backend übergeben. Es wird auch hier über die Tabs iteriert, wobei zum Default-Tab eines leeren Projektes jeweils so viele Tabs wie nötig zusätzlich erstellt werden. Danach wird über die Konstanten der *GlobalDefinitions* die entsprechenden Widgets erstellt und die Felder, die das Widget besitzt, initialisiert. Ähnlich wie bei der *layoutToArray()*-Funktion kann hier ausgenutzt werden, das nicht vorhandene Felder *undefined* sind und als *false* evaluiert werden.

GlobalDefinitions

Das Modul dient als Utility-Objekt und wird als Singleton erstellt. Es sind Felder und Funktionen definiert, die im gesamten Frontend verwendet werden. Unter anderem die Funktionen zum Setzen des *hasLayoutBeenEdited* Feldes, welches dafür genutzt wird, bei Änderungem am Projekt einen Speicherdialog anzuzeigen, wenn die Applikation ohne explizites Speichern geschlossen wird.

Auch sind die verfügbaren Widgets definiert. Das Enum *ComponentType* dient der Identifizierung während der Serialisierung und Deserialisierung. Das Array *componentName* hält die Namen der Widget-Implementierungsdateien und das Array *componentDisplayName* die im Kontextmenü der UI angezeigten Namen der Widgets. Es ist zu beachten, dass die Reihenfolge der Elemente dieser drei Strukturen identisch sein muss, da über die Indezes des Enums auf die Arrays zugegriffen wird.

Zuletzt ist die Funktionen *mapToValueRange* definiert, die einen Eingabewert von einem Ursprungswertebereich in einen Zielwertebereich konvertiert. Diese wird von Widgets wie dem *Slider* verwendet.

Widgets

Ein Widget besitzt mindestens mindestens die Felder: * eventName * label * componentType * componentColor * fontColor

Diese Felder werden vom Widget-Kontextmenü im Allgemeinen Reiter verwendet und sind nicht optional. Soll einem Widget ein Script für die Ausgabe oder Eingabe bereitgestellt werden, so sind entsprechend der Verwendung das Feld *outputScript* und/oder *inputScript* hinzuzufügen. Die Reiter

im Kontextmenü werden ausgegraut, sofern die Felder nicht vorhanden sind.

Zusätzlich kann ein Widget noch ein dediziertes Menü besitzen. Dieses wird im vierten Reiter des Widget-Kontextmenüs, sofern vorhanden, dargestellt.

Widget-Kontextmenü

Das Widget-Kontextmenü besitzt als Wurzelobjekt ein Layout-Objekt, wie dem *GridLayout*. Zudem muss das Feld *var component* im Wurzelobjekt definiert sein. Über dieses Feld kann auf das Widget und dessen Felder zugegriffen werden.

Zum Laden eines Menüs wird die Funktion *loadComponentMenu()* des *EditMenuDialog*-Moduls verwendet. Dort muss für ein neues Menü, ein entsprechender Eintrag erfolgen.

Das Laden des Modul selbst erfolgt über das *Loader*-Objekt im *EditMenuDialog*-Modul. Dieses *Loader*-Objekt wird in einigen Menüs zum dynamischen Laden von Modulen verwendet.

Widget Manipulatoren

Diese befinden sich an den Ecken eines Widgets. Derzeit sind folgende Manipulatoren vorhanden:

- Widget löschen
- Widget skalieren
- Widget drehen
- Widget Kontextmenü

Um einen Manipulator in ein Widget zu integrieren, muss lediglich der Manipulator im Wurzelobjekt des Widgets deklariert werden (siehe Bsp. Widget Manipulator).

Beispiel 3. Widget Manipulator

```
DeleteComponentKnob{
    root: myWidgetRoot
    enabled: myWidgetRoot.enabled
}
```

DebugPopup

Das DebugPopup oder einfach Debugfenster nutzt zur Darstellung der Informationen das *TextArea* -Objekt. Dieses hat die Möglichkeit *RichText*, also formattierten Text darzustellen. Hierzu steht eine Teilmenge des HTML-Sprachumfangs zur Verfügung. In der Funktion *createDebugText()* wird mit Hilfe einer HTML Tabelle ein Eintrag für die *TextArea* erzeugt.

Backend

Die grundlegende Struktur des Backends ist in Abb. Abb. 7 dargestellt. Eine detaillierte Erläuterung erfolgt in den entsprechenden Sektionen.

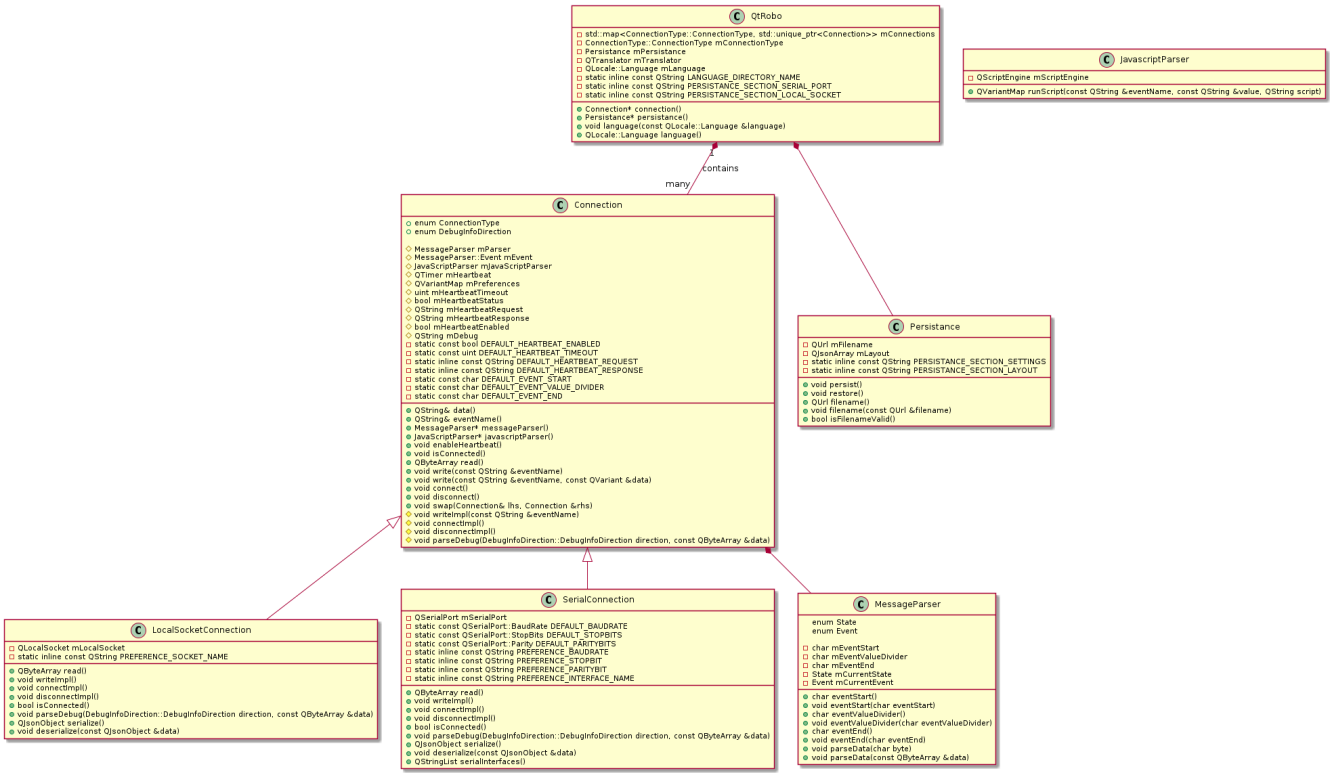

Abbildung 7. Übersicht Backend

Main

Die *main* Fun dient der Initialisierung der Applikation. So wird unter anderem der Theme *Material* gesetzt. Des weiteren wird die Klassen *QSerialPort* als auch *QLocale* für eine Verwendung in QML als Typen registriert (fgl. Bsp. 3).

Beispiel 4. Typregistrierung

```
qmlRegisterType<MyType>("Type", Major, Minor, "QML Type Name");
qmlRegisterUncreatableType<MyUncreatableType>("Type", Major, Minor, "QML Type Name",
"Why not creatable");
```

Zusätzlich werden die in der Klasse *Connection* (siehe Absch. Connection) befindlichen Enums zur Verwendung im Frontend registriert.

Zuletzt wird das Backend *QtRobo* zur Nutzung im Frontend bereitgestellt (fgl. Bsp. 4).

Beispiel 5. QtRobo Registrierung

```
engine.rootContext()->setContextProperty("qtRobo", &qtRobo);
```

Im Anschluss wird die Einstiegsdatei des Frontends *main.qml* (siehe Absch. main.qml) geladen und der Event Loop gestartet.

QtRobo

QtRobo ist die Hauptklasse des Backends. Jegliche Kommunikation zwischen Frontend und Backend

erfolgt hierüber. In einer Map werden alle möglichen Verbindungstypen (derzeit Seriell und Local Socket) als *std::unique_ptr* gehalten, um das Speichermanagment zu automatisieren. Die aktuell ausgewählte Verbindung wird als Enum-Member gehalten. Über die Methode *connection()* wird danach ein roher Pointer an QML übergeben, da QT keine Smartpointer an das QML Frontend verfügbar machen kann. Da auch die *Persistance* Klasse zum Speichern und Laden von Projekten in QML verfügbar sein muss (siehe Absch. Frontend), wird auch diese als roher Pointer an das Frontend übergeben. Verfügbar werden Member oder Getter-/Setter-Methoden über das *Q_PROPERTY* Makro (fgl. Bsp. 5).

Beispiel 6. QT Q_PROPERTY

```
Q_PROPERTY(Typ qmlName READ getterMethod WRITE setterMethod NOTIFY changeSignal)
Q_PROPERTY(Typ qmlName MEMBER classMember NOTIFY changeSignal)
```

Einzelne Methoden, die in QML nutzbar sein sollen, werden mit dem Makro *Q_INVOKABLE* entsprechend registriert (fgl. Bsp. 6).

Beispiel 7. QT Q_INVOKABLE

```
Q_INVOKABLE void restore();
```

Bei einem Speicher- oder Ladevorgang emittiert die Klasse *Persistance* die Signale *serializeConnection* bzw. *deserializeConnection*. Diese sind wiederum mit den Slots *onPersisting* bzw. *onRestoring* der *QtRobo* Klasse verbunden. Innerhalb dieser Slots werden die entsprechenden Methoden der abstrakten Klasse *Connection* zur Serialisierung und Deserialisierung aufgerufen.

Die Klasse ist auch für das Laden der korrekten Übersetzung zuständig. Hierzu wird bei der Initialisierung die Systemsprache genommen und versucht, die korrespondierende Übersetzung zu laden. Ist keine Übersetzung verfügbar, wird auf die Default-Übersetzung (englisch) zurückgegriffen.

Übersetzungseinheiten

Übersetzungen werden mit Hilfe Tools *QT Linguist* erstellt. Dazu werden zuerst eventuell neu hinzugekommene übersetzbare Strings (siehe Frontend) zuerst ermittelt (fgl. Bsp. 7).

Beispiel 8. Update der Übersetzungseinheiten

```
lupdate QtRobo.pro
```

Danach können über *QT Linguist* die Übersetzungen mit der Endung *.ts* (XML Format) bearbeitet werden. Damit die Applikation die Übersetzungen laden kann, müssen diese noch in das entsprechende Arbeitsformat *.qm* (Binärformat) gebracht werden (fgl. Bsp. 8).

Beispiel 9. Erzeugen der Übersetzungseinheiten für QT

```
lrelease de-DE.ts
lrelease en-EN.ts
```

Connection

Die *Connection* Klasse ist abstrakt und definiert die gemeinsamen Schnittstellen einer jeden spezifischen Verbindung. Die Klasse hält den JavaScriptParser (siehe Absch. JavaScriptParser), der vom Frontend genutzt wird. Auch ist der Heartbeat-Mechanismus in dieser Klasse implementiert. Der Heartbeat baut hierbei auf dem *QTimer* auf und wird, sofern aktiviert, bei einem Verbindungsaufbau über die *connect()* Methode gestartet. Nach Ablauf des definierten Intervals sendet der *QTimer* ein *timeout()* Signal aus, welches vom Slot *onHeartbeatTriggered()* aufgefangen wird. Dieser Slot emittet den aktuellen Heartbeat-Status und setzt diesen danach auf *false*. Zuletzt wird eine neue Anfrage an den verbundenen Microcontroller / Middleware gesendet. Der Status des Heartbeat wird nur dann auf *true* gesetzt, wenn eine entsprechende Rückmeldung kam und diese der zuvor definierten Antwort entspricht. Deaktiviert wird der *QTimer* über die *disconnect()* Methode.

Die *Connection*-Klasse abstrahiert so viele Aufgaben wie möglich, sodass die Implementierungen lediglich folgende Methode zu implementieren haben.

- bool isConnected()
- QByteArray read()
- void writeImpl(const QString &eventName)
- void connectImpl()
- void disconnectImpl()
- void parseDebug(DebugInfoDirection::DebugInfoDirection direction, const QByteArray &data)

Bei den Implementierungen handelt es sich dabei um die spezifischen Anforderungen, die die einzelnen Verbindungsarten besitzen. Also wie eine Verbindung aufgebaut wird oder überprüft wird, ob noch verbunden ist. Die *parseDebug()*-Methode stellt dem Debug Fenster des Frontends die nötigen Informationenen bereit. Da diese eventuell von der jeweiligen Implementierung einer Verbindung abhängig sind, existiert hierfür keine allgmeine Implementierung in der *Connection* -Klasse. Um verbindungsspezifische Konfigurationen in allgemeiner Weise zu halten, wurde hierfür auf die Datenstruktur *QVariantMap* zurückgegriffen. Diese ist auch mit dem QML-Frontend kompatibel. Um die Korrektheit der gesetzten Felder dieser Map hat sich jede Implementierung der *Connection* selbst zu kümmern. Generell werden hierzu die entsprechenden Konfigurationspunkte als Konstanten definiert und dienen danach als *Key* der Map.

Derzeit ist die Struktur der Verbindungsimplementierungen wie in Abb. Abb. 7 aufgebaut. Eine Besonderheit besitzt die Implementierung der seriellen Verbindung *SerialConnection*. Da der Nutzer im Frontend die Möglichkeit besitzt, von allen verfügbaren seriellen Schnittstellen des Rechners auszuwählen, wurde speziell in dieser Klasse eine zusätzliche Methode *serialInterfaces()* angelegt. Diese gibt eine Liste an Schnittstellenbezeichnungen zurück.

Persistance

Die Klasse realisiert das Speichern und Laden von Projekten. Dazu gehört auch das Überprüfen, Anlegen und Laden der Projektdatei. Zur Vereinheitlichung der Aufrufe für das Persistieren, existiert die abstrakte Unterklasse *Perstistable*. Diese ist von zu persistierenden Klassen zu implementieren. Die erzeugte Struktur der Projektdatei ist in Bsp. Bsp. 9 dargestellt.

Beispiel 10. QtRobo Projektdatei Struktur

```
{
    'settings':{
        'localsocket': {
            'socketName': 'mySock.sock'
        }
    },
    'layout':[
        {
            'tabName': 'Layout',
            'tabIndex': 0,
            'content': [
                {
                    'componentType': 2,
                    'eventName': 'event',
                    'x': 200,
                    'y': 100,
                    'width': 100,
                    'height': 100,
                    ...
                }
            ]
        }
    ]
}
```

Die Projektdatei ist in zwei Bereiche unterteilt. *Settings* und *Layout*. Im *Settings*-Bereich werden jegliche Informationen über die konfigurierten Verbindungen, inklusive Heartbeat und Eventinformationen gespeichert. Im *Layout*-Bereich werden Informationen über die Widgets gespeichert (siehe Absch. main.qml).

Der Allgemeine Ablauf am Beispiel des Speicherns stellt sie wie in Abb. Abb. 8 zu sehen dar.

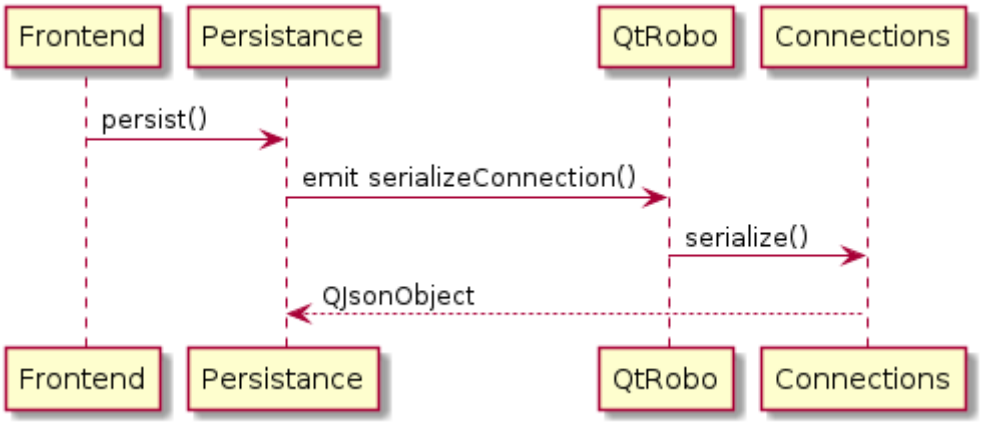

Abbildung 8. Ablauf des Speicherns

MessageParser

Der *MessageParser* verarbeitet eingehende Nachrichten der aktiven Verbindung. Es werden

Nachrichten nur an das Frontend weiter gereicht (siehe Abb. 9), wenn diese den korrekten Aufbau besitzen. Ungültige Zeichen werden je nach Zustand des Parsers entsprechend ignoriert. Verfügbar in QML sind hierbei die Attribute *eventStart*, *eventValueDivider* und *eventEnd*.

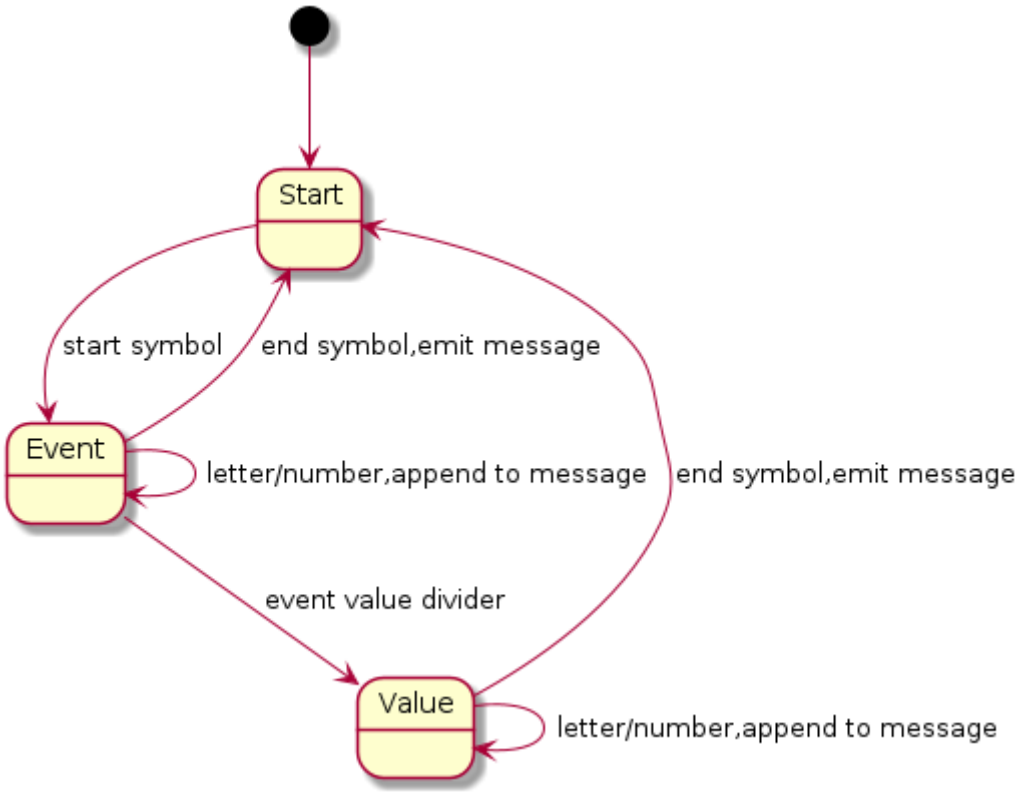

Abbildung 9. Parsen von Nachrichten

JavaScriptParser

Diese Klasse stellt dem Frontend die Methode *runScript(const QString &eventName, const QString &value, QString script) zum Interpretieren von JavaScript-Code zur Verfügung. Zunächst wird das Script auf Korrektheit der Syntax überprüft. Bei korrekter Syntax werden zwei globale Objekte, für den _Eventnamen* und den *Wert*, in der Script Engine gesetzt. Diese sind im Script verfügbar, können ausgelesen und editiert werden. Nachdem das Script interpretiert wurde, werden die aktuellen Werte der beiden globalen Objekte ausgelesen und in einer *QVariantMap* zurückgegeben. Das Frontend kann diese dann entsprechend weiter verarbeiten.

Android Build

QTRobo lässt sich auch als **.apk** Applikation für Android Geräte bauen. Das Bauen der Applikation geschieht hierbei am Einfachsten in einer Windows (10) Umgebung, zum Beispiel in Form einer VM.

Hierzu muss zum einen der QT Online Installer für Open Source Zwecke heruntergeladen und das QT Framework mit folgenden zusätzlichen Modulen installiert werden:

- Android
- Qt Charts
- Qt Script (deprecated)

image::images/TODO

Zum Qt Framework / Creator muss noch (zumindest ist dies die unkomplizierteste Variante) Android Studio installiert werden. Mit Android Studio installieren sich die benötigte Android SDK, Android Build Tools (Gradle etc.) und eine entsprechende Java Version. Diese Komponenten benötigt der QT Creator, um nachfolgend initial das Android Kit zu konfigurieren.

Nach der Installation von Android Studio kann nun im letzten Schritt Android als Zielplattform im QT Creator konfiguriert werden.

Die Konfiguration findet sich unter `Extras → Geräte → Android`. Im Normalfall sollte QtCreator in diesem Tab die Konfiguration automatisch anbieten. Es müssen nur die entsprechenden Lizenzvereinbarungen etc. angenommen werden.

image::images/TODO

Danach kann Qt Robo als Projekt im QT Creator geöffnet werden und es sollte ein entsprechendes Target Kit für Android angeboten werden.

Bibliografie

- [1] RoboRemo, https://www.roboremo.com/
- [2] Qt Framework, https://www.qt.io/
- [3] QML Dokumentation, https://doc.qt.io/qt-5/qtqml-index.html