Developing Toon apps

By

Schelte Bron

July 2019

Table of Contents

1 Introduction

This document describes the development of custom apps for the Toon. It is geared toward development on a Toon that runs firmware version 5. Basic knowledge of javascript and vi is assumed of the reader.

In this document things are sometimes done differently from existing apps. This choice was made to assist in understanding how things really work and prevent cargo cult programming.

2 Widgets

When developing an app for Toon, the need to display controls and information on the screen will quickly arise. The way to do this is via visual items, also commonly known as widgets. To get started, a few basic widgets must be introduced.

2.1 Item type

The Item type is the base type for all visual items. All visual items inherit from Item. The Item component is defined in QtQuick. So, to use any visual item, always import QtQuick in the qml file. An Item object has no visual appearance, but it defines all the attributes that are common across visual items. The Item type is very helpful for grouping visual items.

Some useful properties of an Item are: opacity, rotation, visible. The Item object also defines properties, used for widget placement. This is discussed in detail in paragraph 3.2.

2.2 Text type

A Text Item is used to display text on the screen. The most important attribute of a Text item is text, which defines the actual text string to display. The appearance can be controlled through the font group and the color attribute. The text can be aligned using the horizontalAlignment (Text.AlignLeft, Text.AlignRight, Text.AlignHCenter, Text.AlignJustify) and verticalAlignment (Text.AlignTop, Text.AlignBottom, Text.AlignVCenter) attributes.

The most important attributes in the font group are font.family and font.pointSize. Many existing apps use font.pixelSize, but using font.pointSize makes the font device independent, so the app will render appropriately on both Toon 1 and Toon 2.

2.3 Image type

An Image item shows the image specified via the source property. The value of the source property should be a url. As shown before, the qrc scheme can be used for images stored in a local resources file, like drawables.rcc or resources.rcc. The http scheme is available for images on the network. While it is unfortunately not possible to use images stored in files directly, they can be obtained via http by placing them somewhere under /HCBv2/qml/images.

For example, an image called icon.png placed in /HCBv2/qml/images/demo can be accessed as http://localhost/images/demo/icon.png. To keep the images in the app directory, they can be placed in a drawables directory as was customary in earlier versions of the Toon firmware. Then create a symbolic link under /HCBv2/qml/images. Example:

```
cd /HCBv2/qml/images
ln -s ../apps/demo/drawables demo
```

Similar to Text objects, Image objects can also be aligned using the horizontalAlignment (Image.AlignLeft, Image.AlignRight, Image.AlignHCenter) and verticalAlignment (Image.AlignTop, Image.AlignBottom, Image.AlignVCenter) attributes.

Some other useful properties of an Image object are: cache, fillMode.

2.4 Rectangle type

The Rectangle item is frequently used to fill areas with solid color or gradients, and/or to provide a rectangular border. It provides the following properties:

antialiasing : bool

border.width : int

border.color : color

color : color

gradient : any

radius : real

3 Layout tools

It is not enough to have a bunch of widgets. There must also be a way to control where the widgets are placed on the screen. This can be done using absolute positioning, the anchors property group, or with layout items. Different items can use different methods to get the desired result, but each item should use only one method.

3.1 Absolute positioning

Each visual item inherits the x, y, width, and height properties from the Item object. The x and y properties indicate the position of an item within its parent. The top left-hand corner of the parent corresponds to (0, 0). The x and y values govern where the top left-hand corner of the item will be placed.

The size of the item is specified using the width and height properties.

3.2 Anchors

Each visual item inherits the anchors property group from the Item object. Anchors allow items to be placed relative to their parent or siblings.

3.2.1 Anchor lines

Each item has 7 anchor lines; 4 horizontal (top, verticalCenter, bottom, baseline) and 3 vertical (left, horizontalCenter, right).

Anchor lines of an item can be attached to an anchor line of another item. For example, by setting the horizontalCenter anchor of an item to the horizontalCenter anchor of the parent, the item will be horizontally centered in its parent:

```
anchors.horizontalCenter: parent.horizontalCenter
```

To put an item below its sibling with id "topBar", use:

```
anchors.top: topBar.bottom
```

For convenience, the fill and centerIn properties are available. The fill property sets the top, bottom, left, and right anchor lines to those of the specified item. The centerIn property does the same for the horizontalCenter and verticalCenter anchor lines.

The baseline anchor line corresponds to the imaginary line on which text would sit. For items with no text it is the same as top.

3.2.2 Anchor margins and offsets

It is not always desirable for an anchor line of one item to be exactly the same as the anchor line of another item. There may be a need to leave some space between the items. This can be accomplished using margins. The topMargin, bottomMargin, leftMargin, and rightMargin properties are used to define how much space is required on each side. The margins property can be used to set all four to the same amount. Positive values insert space around the item. Negative values make the item overlap the anchor line specified as the reference.

For the other three anchor lines, offsets are used; verticalCenterOffset, baselineOffset, and horizontalCenterOffset. Positive values move the item down or to the right of the reference anchor line. Negative values move it up or to the left.

3.3 Layout items

3.3.1 Column

Column is a type that positions its child items along a single column. It can be used as a convenient way to vertically position a series of items without using anchors. Use the spacing property to set the amount in pixels left empty between adjacent items.

```
Screen {
    Column {
        spacing: 5
        Text {
            text: "Some text"
        }
        Text {
            text: "Next line"
        }
    }
}
```

3.3.2 Row

Row is similar to Column, except that it positions its child items along a single row. It can be used as a convenient way to horizontally position a series of items without using anchors. Use the spacing property to set the amount in pixels left empty between adjacent items.

3.3.3 Grid

A Grid creates a grid of cells that is large enough to hold all of its child items, and places these items in the cells from left to right and top to bottom. Each item is positioned at the top-left corner of its cell with position (0, 0).

A Grid defaults to four columns, and creates as many rows as are necessary to fit all of its child items. The number of rows and columns can be constrained by setting the rows and columns properties.

3.3.4 Repeater

The Repeater type is used to create a large number of similar items. A Repeater has a model and a delegate: for each entry in the model, the delegate is instantiated in a context seeded with data from the model. A Repeater item is usually enclosed in a positioner type such as Row or Column to visually position the multiple delegate items created by the Repeater.

There should only be one item inside the Repeater. Of course this can be an Item object, containing multiple other items. The full size of the enclosed item must be clear to the Repeater item. For things like Text items that is the case without extra work, but for an Item object, it must be specified. Even if one dimension seems irrelevant, like the width in a Column object, both width and height must be specified. Rather than hard-coding these values, properties from the childrenRect group can be used. The childrenRect property holds the collective position and size of the item's children.

Here's an example screen that shows a list of all enabled apps:

```
import QtQuick 2.1
import qb.components 1.0

Screen {
    Grid {
```

```
        anchors.fill: parent
        anchors.leftMargin: 30
        flow: Grid.TopToBottom
        rows: Math.ceil(globals.enabledApps.length / 3)
        Repeater {
            model: globals.enabledApps
            Row {
                width: childrenRect.width
                height: childrenRect.height
                spacing: 5
                Text {
                    width: 20
                    text: index
                    horizontalAlignment: Text.AlignRight
                }
                Text {
                    width: 220
                    text: modelData
                }
            }
        }
    }
}
```

The model property can either be a number, or an array. A number simply indicates the amount of delegates to be created by the repeater. When an array is used, the number of delegates that are created is equal to the number of elements in the array. For each delegate the corresponding array element is available via the modelData property. In either case, the index property holds the index of the delegate within the repeater.

4 Demo app

4.1 Files and directories

For an app to be picked up by the system, it must adhere to the following rules:

- The code is placed in a directory under /HCBv2/qml/apps with the name of the app, starting with a lowercase letter.
- The app directory must contain a file called *Appname*App.qml, where *Appname* is the same as the directory name with only the first letter changed to uppercase.

You may see other files and directories in existing apps, like qmldir and drawables, but they are just relics of older QtQuick versions.

4.2 DemoApp.qml

The only required file in an app will normally define an App component. The init() function declared inside the App component is called when the GUI starts. It normally registers a number of widgets. Common widgets include "tile", "screen", and "systrayIcon".

As an example, let's make an app called "demo". The first step is to create a directory for the app and then go into that directory:

```
cd /HCBv2/qml/apps
mkdir demo
cd demo
```

Now create a file called DemoApp.qml using vi. Put the following code in the file:

```
import QtQuick 2.1
import qb.base 1.0

App {
    property url tileUrl: "DemoTile.qml"

    function init() {
        const args = {
            thumbCategory: "general",
            thumbLabel: "Demo",
            thumbIcon: "qrc:/apps/clock/drawables/clock.svg",
            thumbIconVAlignment: "center",
            thumbWeight: 30
        }
        registry.registerWidget("tile", tileUrl, this, null, args);
    }
}
```

Restart the GUI by killing the qt-gui process: `killall qt-gui`

When the GUI returns, you will now find a "Demo" tile in the general section of the tile selection screen. Obviously it is not yet possible to actually add the tile because there is no code for the tile at this moment. That is covered in the next paragraph.

The registerWidget() function takes up to 5 arguments:

typeName:

> The type of widget to register.

widgetUrl:

> The URL containing the definition of the widget. Instead of just passing a string, a url property should be used in order for the framework to compile it into a usable form.

appContext:

A reference to the app the widget belongs to. This can be the id of the app, if defined, or simply *this*, because the widget is registered from within the app.

localName:

The name of the variable in the app that should be updated to point to the latest instantiated widget of this type or null if not used.

args:

Arguments to be passed to the widget initialisation. For tile widgets these are mainly used to control what the thumbnail will look like.

4.3 DemoTile.qml

To create a tile, just make a file with the name that was indicated in the registerWidget() call. But this time a Tile should be defined, instead of an App.

```
import QtQuick 2.1
import qb.components 1.0

Tile {
    Text {
        text: "This is a demo"
    }
}
```

Of course, that doesn't look very nice, but it works. You will probably want to adjust the font and placement of the text. But this was just an example of the bare minimum needed for creating a tile.

4.4 DemoScreen.qml

The definition of a screen is pretty much the same as a tile, except for using a Screen component, instead of a Tile component. However, registering it from inside the app is a bit different.

First of all, another url property must be defined so the framework will compile the source file that defines the screen. Then you need another property that will hold a reference to the widget once it is instantiated. The type of this property is the name of the screen object.

Finally, the screen must be registered. This is done inside the init() function of the app. Screens don't need to provide any value for the args argument of registerWidget(), but the name of the screen object property must be passed as the localName argument. That way, the property will automatically be set to the screen widget instance.

With these changes, DemoApp.qml looks like this:

```
import QtQuick 2.1
import qb.base 1.0

App {
    property url tileUrl: "DemoTile.qml"
    property url screenUrl: "DemoScreen.qml"

    property DemoScreen demoScreen

    function init() {
        const args = {
            thumbCategory: "general",
            thumbLabel: "Demo",
            thumbIcon: "qrc:/apps/clock/drawables/clock.svg",
            thumbIconVAlignment: "center",
            thumbWeight: 5
        }
        registry.registerWidget("tile", tileUrl, this, null, args);
        registry.registerWidget("screen", screenUrl, this, "demoScreen");
    }
}
```

To be able to actually access the screen, it can be attached to clicking the tile. That is accomplished by calling the show() method of the screen from the onClicked handler of the tile. Like this:

```
import QtQuick 2.1
import qb.components 1.0

Tile {
    onClicked: {
        if (app.demoScreen) app.demoScreen.show()
    }

    Text {
        text: "Click here"
        font {
            family: qfont.bold.name
            pointSize: 18
        }
        anchors {
            top: parent.top
            topMargin: 20
            horizontalCenter: parent.horizontalCenter
        }
    }
}
```

This uses the demoScreen property from the app. For good measure a check is done that the property has been set. In other words, check that the screen instantiation has finished.

4.5 DemoTray.qml

Another possibility to access the screen is via a system tray icon. Once again a url property pointing to the source file implementing the tray icon must be defined in the app. The icon must also be registerd in the init() function of the app. For a system tray icon, both the localName and the args arguments can be omitted. So, assuming the url property is called trayUrl, the widget registration should look something like this:

```
registry.registerWidget("systrayIcon", trayUrl, this);
```

The implementation of the system tray icon uses the SystrayIcon component. The item must have an id, and a string property called objectName containing that id. Normally the item will also have an onClicked handler and an Image component, centered in the available space.

```
import QtQuick 2.1
import qb.components 1.0

SystrayIcon {
    id: demoSystrayIcon
    property string objectName: "demoSystrayIcon"

    onClicked: {
        if (app.demoScreen) app.demoScreen.show()
    }

    Image {
        anchors.centerIn: parent
        source: "qrc:/apps/clock/drawables/clock.svg"
    }
}
```

4.6 Defining custom items

Creating your own items is very simple too. Just create a file that defines the layout of the item with the desired name of the item and a .qml extension. For example, let's recreate the bar graph used by the stock tile that shows the present power consumption. The bar graph is made up of 10 bars of 32x5 pixels, with 3 pixels of space between them. The number of bars that light up represents the amount of power currently being used. The stock implementation is done in a very simple way, requiring the caller to do a large part of the work. Here is a better approach, that demonstrates a few useful principles:

```
import QtQuick 2.1

Item {
    width: childrenRect.width
    height: childrenRect.height

    property int maxValue: 5000
    property int value: 0
    property bool dimState: canvas.dimState
```

```
    Component.onCompleted: redraw(value)
    onValueChanged: redraw(value)
    onMaxValueChanged: redraw(value)
    onDimStateChanged: redraw(value)

    function redraw(val) {
        var count = barGraph.count
        var num = Math.round(val / (maxValue / count));
        for (var i = 0; i < 10; i++) {
            barGraph.itemAt(i).color = i < num ? "#ff0000" : "#aaaaaa"
        }
    }

    Column {
        spacing: 3
        Repeater {
            id: barGraph
            model: 10
            Rectangle {
                width: 32
                height: 5
                color: "#aaaaaa"
            }
        }
        rotation: 180
    }
}
```

The ten bars are created using a Repeater object. An id has been assigned to the Repeater to be able to refer to it from the redraw() function. The Repeater is put into a Column object so the bars will be shown in a neat stack. A Column object puts its items in a top to bottom order, but for a bar graph it would be more convenient to have the items in a bottom to top order. For this reason the column is rotated 180 degrees.

To adjust the display based on the current value, the object has a redraw() function. When invoked, it calculates the number of bars that should be on, and then configures the colors of all bars accordingly.

Two properties have been defined for the BarGraph object: value, representing the current value to be indicated on the graph, and maxValue, which defines the range of the bar graph. When either of those properties changes, the graph must be updated. This is accomplished using the onValueChanged and onMaxValueChanged event handlers that invoke the redraw() function.

To make sure the graph will correctly show the initial value, the redraw() function must also be invoked when the widget is first rendered. This is done via the Component.onCompleted handler. And finally, the graph also has to be redrawn when the dim state changes (see

paragraph 6.1). For this, the dimState property must be pulled in from the canvas object, with a handler applied to it.

The BarGraph can be placed on the tile of the demo app to see it in action. Put the following code into DemoTile.qml and restart the GUI (killall qt-gui). Now every time the tile is pressed a random value for the graph is generated:

```
import QtQuick 2.1
import qb.components 1.0

Tile {
    property int power: 3456

    onClicked: {
        power = Math.round(5000 * Math.random())
    }

    BarGraph {
        value: power
        anchors.centerIn: parent
    }
    Text {
        text: power
        anchors {
            baseline: parent.bottom
            baselineOffset: -20
            horizontalCenter: parent.horizontalCenter
        }
    }
}
```

5 Other objects

5.1 Timer

A Timer can be used to trigger an action either once, or repeatedly at a given interval.

The interval property determines after how many milliseconds the timer will be triggered after it has been started.

Starting and stopping the timer is done either by setting the running property, or using the start() and stop() methods. The restart() method causes the timer to reset and start a new interval.

If the repeat property is true, the timer will automatically restart when the specified time has elapsed. If the repeat property is false, the running property changes to false after the timer has been triggered.

Setting the triggeredOnStart property to true will produce an extra trigger when the timer is started.

When the timer is triggered, the onTriggered handler is invoked.

For example, a timer for a clock could be defined like this:

```
Timer {
    interval: 1000
    triggeredOnStart: true
    running: true
    repeat: true
    onTriggered: updateClock()
}
```

6 Techniques

6.1 Dim state

When no user interaction has taken place for some configurable amount of time, the Toon goes to dim state. Tiles should be designed to adjust their appearance appropriately. As the background switches from light colors to black, many items may blend into the background if no precautions have been taken. The dimState variable indicates whether the display is dimmed. This can be used to hide or reposition certain items, or adjust their appearance.

```
Image {
    source: dimState ? dimImage : stdImage
}

Text {
    text: "This text is hidden when dimmed"
    visible: !dimState
}
```

To simplify the design, the app can use special color definitions that have been prepared to react to changes in dimState. For example, setting the color property of a Text object to colors.tileTextColor will result in dark gray characters, which change to white in dimmed state:

```
Text {
    text: "Hello, World!"
    color: colors.tileTextColor
}
```

The whole list of available colors that adapt to dimmed state can be found in appendix A

6.2 Signals

Code can emit a signal to notify interested parties about events.

When the value of a property changes the on<Property>Changed signal is emitted, where <Property> is the name of the property, with the first letter capitalized.

6.3 XMLHttpRequest

The XMLHttpRequest class can be used to asynchronously obtain data from over a network. Contrary to what the name may suggest, this is not limited to XML only.

The steps to perform a http request are:

1. Create an XMLHttpRequest object
2. Setup a handler for ready state changes
3. The handler should check that the readyState is DONE (4) and the status is OK (200) before processing the responseText
4. Provide the method (GET, POST, PUT) and url to the XMLHttpRequest object
5. Initiate the request

Example:

```
function geturl(url, processFunc) {
    var req = new XMLHttpRequest()
    req.onreadystatechange = function() {
        if (this.readyState == XMLHttpRequest.DONE && this.status == 200) {
            processFunc(this)
        }
    }
    req.open("GET", url)
    req.send()
}
```

The XMLHttpRequest class can also be used to read and write local files, for example to store configuration settings for an app. Reading is done exactly like getting a web page, but with a file:// url. To write a file, use a "PUT" operation on the file url. Then pass the data to be written into the file to the send() method:

```
    var fd = new XMLHttpRequest()
    fd.open("PUT", "file:///etc/demorc")
    fd.send("Some data")
```

6.4 Working with JSON data

Convert JSON data to an object: var obj = JSON.parse(str)

Convert an object to a JSON string: var str = JSON.stringify(obj)

6.5 Working with XML data

The easiest way to process XML data is to use XmlListModel. Using this method, the desired information can be accessed using XPath queries. For example, to get the current weather and temperature for Utrecht from buienradar:

```
import QtQuick 2.1
import qb.components 1.0
import QtQuick.XmlListModel 2.0

Screen {
    XmlListModel {
        id: station
        source: "https://data.buienradar.nl/1.0/feed/xml"
        query: "//weerstations/weerstation[@id='6260']"
        XmlRole {
            name: "temp"
            query: "temperatuurGC/string()"
        }
        XmlRole {
            name: "str"
            query: "icoonactueel/string(@zin)"
        }
    }

    ListView {
        anchors.fill: parent
        anchors.leftMargin: 30
        model: station
        delegate: Text {
            text: str + ", " + temp + " graden"
        }
    }
}
```

If the extracted data is wanted for further processing, instead of directly being displayed, it can be accessed using the get() method of the model. In that case, care must be taken that the XML data processing has completed before the method is invoked. The model has a status property that changes to XmlListModel.Ready when the data processing is complete. However, there is no associated signal. To mitigate that, an alias property can be used.

Here is an example that uses that technique to read Toon's network settings and creates a QR-Code to allow visitors to easily connect to wifi.

```
import QtQuick 2.1
import qb.components 1.0
import BasicUIControls 1.0
import QtQuick.XmlListModel 2.0

Tile {
    property alias progress: config.status

    onProgressChanged: generate()

    function generate() {
        if (config.status == XmlListModel.Ready) {
            qrCode.content = "WIFI:T:WPA;S:" + config.get(0).essid
              + ";P:" + config.get(0).psk + ";;"
        }
```

```
    }

    XmlListModel {
        id: config
        source: "file:///HCBv2/config/config_hcb_netcon.xml"
        query: "/Config/device/interfaceSettings/wifi_essid/.."
        XmlRole {
            name: "essid"
            query: "wifi_essid/string()"
        }
        XmlRole {
            name: "psk"
            query: "wifi_key/string()"
        }
    }

    QrCode {
        id: qrCode
        anchors.centerIn: parent
        width: 100
        height: width
    }
}
```

6.6 WebSockets

When an app wants to display some information that may change at any time, the classic method was to poll the information source at regular intervals to check if the data has changed. This is inefficient, because for certain types of information, the data usually doesn't change. In that case the poll will result in getting the same data many times in a row. Then when the data does change, there's an average delay of half the poll interval before the change is known to the app. A much better approach is to have the information source push updates, as soon as they happen. This can be achieved with websockets:

```
import QtQuick 2.1
import qb.components 1.0
import QtWebSockets 1.1

Screen {
    onShown: socket.active = true
    onHidden: socket.active = false

    WebSocket {
        id: socket
        url: "ws://echo.websocket.org"
        onTextMessageReceived: {
            messageBox.text = messageBox.text + "\nReceived: " + message
        }
        onStatusChanged: {
            if (this.status == WebSocket.Error) {
                console.log("Error: " + socket.errorString)
```

```
            } else if (this.status == WebSocket.Open) {
                this.sendTextMessage("Hello World!")
            } else if (this.status == WebSocket.Closed) {
                messageBox.text += "\nSocket closed"
            }
        }
        active: false
    }
    Text {
        id: messageBox
        text: socket.status == WebSocket.Open ? "Sending..." : "Welcome!"
        anchors.centerIn: parent
    }
}
```

When the screen is opened by the user, the websocket connection is initiated via the screen's onShown handler. The onTextMessageReceived handler of the websocket is invoked every time a message is received. When the user navigates away from the screen, the onHidden handler tears down the websocket again.

6.7 Configuration files

If an app needs to save some configuration settings, it may be tempting to store those in a file in the app directory. However, that means that when a new version of the app is installed, the configuration file must be copied to the new directory to avoid losing the settings. In my opinion it is better to store the configuration in a file under /HCBv2/qml/config. Although in that case care must be taken not cause a conflict with another app. So definitely don’t call the file config.json.

The easiest format to work with for configuration files is json. That format can simply be converted to and from an object.

Note: When performing a GET on a non-existing local file, the returned status and statusText end up to be 0 and "", not 404 and "Not Found", as could reasonably be expected.

7 BoxTalk

BoxTalk is the protocol that Toon uses for inter-process communication. It appears to make use of ssdp for device discovery.

An app that wants to communicate with another BoxTalk process would normally import the BxtClient module, set up a BxtDiscoveryHandler and one or more BxtDatasetHandlers. Using the BxtDiscoveryHandler, the UUID of the other process can be discovered, which is needed to send messages to that process. The BxtDatasetHandlers handle incoming messages from the remote process.

To send a message to the remote process, the first thing to do is to create a message object, using bxtFactory.newBxtMessage(). This function takes 4 arguments: The type of message (BxtMessage.ACTION_INVOKE or BxtMessage.ACTION_RESPONSE), the UUID of the remote process, the service class, and the operation to perform. Next, arguments may be added to the message via the addArgument() method. Finally the message is sent using bxtClient.sendMsg() or bxtClient.doAsyncBxtRequest(). If the result of an action needs to be collected when using bxtClient.sendMsg(), a BxtResponseHandler must be defined.

Which services and operations are available depends on the remote process. This information can possibly be obtained via an ssdp query. This needs further investigation.

The following example shows the current temperature reported by happ_thermstat:

```
import QtQuick 2.1
import qb.components 1.0
import BxtClient 1.0

Screen {
    property string temperature: "unk"
    property string thermostatUuid

    Column {
        anchors.fill: parent
        anchors.leftMargin: 30
        Text {
            text: thermostatUuid
        }
        Text {
            text: temperature
        }
    }

    function thermostatInfoParse(node) {
        temperature = node.getChild("currentTemp").text
    }

    BxtDiscoveryHandler {
        id: thermstatDiscoHandler
        deviceType: "happ_thermstat"
        onDiscoReceived: {
            thermostatUuid = deviceUuid;
        }
    }

    BxtDatasetHandler {
        dataset: "thermostatInfo"
        discoHandler: thermstatDiscoHandler
        onDatasetUpdate: thermostatInfoParse(update)
    }
}
```

8 Internationalization

To use text that can be presented in the language of choice of the user, the qsTr() function would normally be used. If the string passed to qsTr() is found in a language mapping for the currently selected language, qsTr() uses the translated string. Otherwise, it uses the string that was passed directly. Next, qsTr() replaces occurrences of %*n* with the *n*-th parameter specified via the arg() function. The resulting string is then returned.

```
text: qsTr("Error code %1").arg(errCode)
```

The translations of strings used in apps are normally stored in files in a lang subdirectory of the app directory. The translations are created in files named lang_<language>_<country>.ts, where <language> is a two-letter lowercase code for the language, and <country> a two-letter uppercase code for the country where a variant of that language may be spoken.

A language file that provides translation to Dutch for a demo app might look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS>
<TS version="2.1" language="nl_NL" sourcelanguage="nl">
<context>
    <name>DemoApp</name>
    <message>
        <source>Demo</source>
        <translation>Voorbeeld</translation>
    </message>
</context>
</TS>
```

The name node refers to the qml file that uses the strings defined in message nodes within the context. Multiple message nodes may be used to define multiple stings. Multiple context nodes may be defined to provide translations for different files of the app.

When this file has been created or modified, it can be converted to a binary qm file using the lrelease tool, which is available on Toon:

```
lrelease lang_nl_NL.ts
```

If code in one file of the app wants to use a string that was already translated for use in another file, the qsTranslate() function can be used to avoid having to duplicate the translation. The context of the other file then needs to be passed as the first argument to the qsTranslate() function:

```
qsTranslate("DemoScreen", "Some string")
```

At the moment, translations for custom apps are not automatically loaded. Hopefully this will be corrected by the TSC in a future update. As a work-around, the translations can be loaded by adding the loadLanguagePackage() function to the app's init() function. E.g.:

```
qlanguage.loadLanguagePackage("apps/demo/lang")
```

In addition to text, also numbers and dates may need to be localized. Some countries use a period as the decimal separator, while a comma is used in other places. The order that the year, month, and day are normally listed is also very country dependent. To handle these cases, several i18n functions are available:

i18n.decimalSeparator()

i18n.number(value, precision, options, arg)

Available options are:

- i18n.general_rounding (arg = rounding point)
- i18n.omit_trail_zeros

i18n.dateTime(time, options)

Available options are:

- i18n.date_no / i18n.date_yes
- i18n.year_no / i18n.year_yes
- i18n.mon_short / i18n.mon_full
- i18n.dom_no / i18n.dom_yes
- i18n.dow_short / i18n.dow_full
- i18n.time_no / i18n.time_yes
- i18n.hour_str_no / i18n.hour_str_yes
- i18n.secs_no / i18n.secs_yes
- i18n.leading_0_no / i18n.leading_0_yes

i18n.duration(duration)

i18n.currency(amount, options)

Available options are:

- i18n.curr_round

i18n.capitalizeFirstChar(string)

9 Measurements

The screen of the Toon 2 is roughly 1.25 times the size of the Toon 1. When developing apps that should look good on both versions, you can use the isNxt variable to adjust the used pixel sizes. E.g:

```
    anchors.leftMargin: isNxt ? 25 : 20
```

9.1 Toon 1

Screen size: 800x480

Tile size: 230x158

Top bar icons: 56x56 (normally icons are 24x24 with a 16 pixel margin)

9.2 Toon 2

Screen size: 1024x600

10 Debugging

10.1 Basic debugging

Debugging apps on the Toon is quite difficult. If your app doesn't work, the first thing to do is look at the timestamp of the qmlc files. If they don't exist, or have a time stamp older than the matching qml file, there is syntax error in the qml file.

If the qmlc file exists and has a time stamp that is newer than the qml file, but the app doesn't work, there is a semantic error. To investigate these, some temporary debugging code can be added to the qml file. First, a string property is needed to hold the debugging information. Next, the information can be displayed on the screen using a text item. Then, throughout the code, the string property can be set to show different pieces of information. This way it is possible to locate where things go wrong. By using a try … catch construct, an error message can be obtained.

```
import QtQuick 2.1
import qb.components 1.0

Screen {
    function init() {
        try {
            ...
        } catch (err) {
            debug = err.message
        }
    }

    // Debugging
    property string debug: ""
    Text {
        text: debug
        anchors {
```

```
            left: parent.left
            right: parent.right
            bottom: parent.bottom
            leftMargin: 5
        }
    }
}
```

10.2 Logging

10.2.1 Activate logging

If the naive debugging methods don't work, it is possible to log the output of qt-gui to a file. To do this, first make a backup of /etc/inittab, then change the qtqt line from:

```
qtqt:245:respawn:/usr/bin/startqt >/dev/null 2>&1
```

to:

```
qtqt:245:respawn:/usr/bin/startqt >/var/log/qt 2>&1
```

You will have to tell the system to use the updated inittab:

```
init q
```

The next time the qt-gui is restarted (using `killall qt-gui`), the output of the new instance will go to /var/log/qt.

Warning: Qt may produce quite a lot of output, which will fill up the available space of the /var/log partition if left running too long.

As a precaution, you can already restore inittab to its original version at this point. It won't be used until you tell the system to do so (init q), or at the next boot. This way, if things go haywire and Toon reboots, all is well again.

To send debugging information from your app to the log, you can insert calls to console.log() in your code.

10.2.2 Deactivate logging

When all bugs are fixed, deactivate logging using the following steps:

1. Restore the inittab backup, if not yet done (or replace /var/log/qt by /dev/null)
2. Tell the system to reload inittab: `init q`
3. Restart the qt-gui: `killall qt-gui`
4. Clean up /var/log: `rm /var/log/qt`

Appendix A – Predefined colors

Name	Normal	Dimmed
canvas	#dcdcdc	#000000
topbar	#dcdcdc	#000000
background	#ffffff	#000000
foreground	#565656	#898989
separator1	#dcdcdc	#000000
separator2	#dcdcdc	#000000
contrastBackground	#f0f0f0	#000000
clockTileColor	#565656	#ffffff
menuBarLabel	#000000	#898989
menuLabel	#565656	#ffffff
btnShadow	transparent	transparent
btnUp	#a8a8a8	#ffffff
btnUpPrimary	#e64f0a	#ffffff
btnDown	#565656	#f0f0f0
btnDownPrimary	#cc3300	#f0f0f0
btnSelected	#ffffff	#ffffff
btnDisabled	#f0f0f0	#a8a8a8
btnText	#ffffff	#000000
btnTextPrimary	#ffffff	#000000
btnTextDown	#ffffff	#000000
btnTextSelected	#e64f0a	#000000
btnTextDisabled	#a8a8a8	#000000

Name	Normal	Dimmed
onOffToggleLeft	#a8a8a8	#dcdcdc
onOffToggleLeftShadow	#565656	#b4b4b4
onOffToggleRight	#388e3c	#9f9f9f
onOffToggleRightShadow	#00695c	#898989
spText	#565656	#ffffff
spBackgroundValue	#ffffff	#000000
spBackgroundButtonsUp	#ffffff	#000000
spBackgroundButtonsDown	#a8a8a8	#171717
spOverlayButtonsUp	#565656	#898989
spErrorText	#cc3300	#ffffff
tempTileTextUp	#565656	#898989
tempTileTextDown	#e64f0a	#ffffff
tempTileBackgroundUp	#ffffff	#000000
tempTileBackgroundDown	#ffffff*0.7	#000000
tempTileBorderUp	transparent	#898989
tempTileBorderDown	transparent	#000000
waTileTitleTextColor	#565656	#000000
waTileTextColor	#565656	#ffffff
graphTileRectText	#ffffff	#000000
graphTileRect	#8e24aa	#ffffff
dayTileAverageBar	#8e24aa	#ffffff
dayTileUsageBar	#512da8	#ffffff
dayTileMiddleBar	#ffffff	#000000
dayTileFixedCostBar	#cccc33	#b4b4b4

Name	Normal	Dimmed
dayTileBackgroundBar	#dcdcdc	#565656
tileTitleColor	#565656	#000000
tileTextColor	#565656	#ffffff
graphSolarThisMomentDot	#ff9900	#ffffff
graphElecSingleOrLowTariff	#8e24aa	#ffffff
graphElecSingleOrLowTariffSelected	#512da8	#ffffff
graphElecHighTariff	#e91e63	#ffffff
graphElecHighTariffSelected	#a6115b	#ffffff
graphGasDistrictHeat	#00afbf	#ffffff
graphGasDistrictHeatSelected	#0097a7	#ffffff
graphWater	#6699ff	#ffffff
graphWaterSelected	#295fcc	#ffffff
graphSolar	#ff9900	#ffffff
graphSolarSelected	#ff6600	#ffffff
powerTileBar0	#00695c	#ffffff
powerTileBar1	#388e3c	#ffffff
powerTileBar2	#689f38	#ffffff
powerTileBar3	#cccc33	#ffffff
powerTileBar4	#ffcc33	#ffffff
powerTileBar5	#ff9900	#ffffff
powerTileBar6	#ff6600	#ffffff
powerTileBar7	#e64f0a	#ffffff
powerTileBar8	#cc3300	#ffffff
powerTileBar9	#cc3300	#ffffff

Name	Normal	Dimmed
powerTileBarEmpty	#dcdcdc	#565656
takeAndReturnDotTaking	#8e24aa	#ffffff
takeAndReturnDotFeeding	#ff9900	#ffffff
notificationsText	#000000	#000000
notificationsTextHighlight	#e64f0a	#000000
notificationsBackground	#dcdcdc	#000000

Appendix B – Modules

B.1 QtQuick

The Qt Quick module is the standard library for writing QML applications. It should normally be imported into all qml files. The Qt Quick module provides all the basic types necessary for creating user interfaces with QML. It provides a visual canvas and includes types for creating and animating visual components, receiving user input, creating data models and views and delayed object instantiation. For more information, see the QtQuick documentation.

B.2 qb.base

This module provides a few basic components:

B.2.1 App

Base functionality for any App. Handles setting of automagic references to instantiated widgets.

B.2.2 Widget

Provides the base functionality for any widget.

Sets references from Widget to owner App and back. When everything is set-up init() will be invoked.

B.3 qb.components

This module provides a whole lot of useful components to build a user interface:

B.3.1 AdvicePopup

B.3.2 AlphaNumericKeyboard

B.3.3 AreaGraph

Component extending AreaGraphControl with other features such as X legend representing 24 hours, Y legend values, horizontal guiding lines, warning icon when there are missing (NaN) values.

Y legend values starts at @maxValue and drops to 0 by fixed delta (dependant on @yLegendItemCount).

X legend values represents hours from 0:00 till 24:00 with visible only each 2nd hour.

B.3.4 BarButton

B.3.5 BottomTabBar

B.3.6 BottomTabButton

B.3.7 CategoryListItem

Used to select the different frames from a vertical menu

See an example usage in the settings app. Every CategoryListItem opens a different frame in the settings screen.

B.3.8 DashedLine

B.3.9 DateSelector

B.3.10 DateSpinner

B.3.11 DescriptiveRadioButton

A custom delegate component for radiobutton list.

In a group of radio buttons, only one radio button can be checked at a time. If the user selects another button, the previously selected button is switched off.

B.3.12 DialogPopup

Footer container, only visible when one off the two buttons contains text. If not visible set height to 0. This will center dialog content horizontaly nice in dialog (not taking buttons area into account).

B.3.13 DottedSelectorDot

B.3.14 DottedSelector

B.3.15 DoubleLabel

B.3.16 DoubleLineH

B.3.17 DoubleLineV

B.3.18 DynamicTopTabButton

B.3.19 EditTextLabel

B.3.20 ErrorButton

B.3.21 ErrorCard

B.3.22 ErrorCardsView

B.3.23 FilterableListAlphaKeyboard

B.3.24 FilterableListDelegate

B.3.25 FilterableListImplementation

Component displaying up to fixed number of items scrollable by up/down buttons. Scrolling is page based and is always scrolled to have the itemsPerPage-th item at the top of the visible items - might be empty lines at the last page when total count is not itemsPerPage multiple.

Using two data models. One model for all available items - dataModel, out of which only fixed number (itemsPerPage) of items are displayed using a Repeater and using the second model - repeaterModel.

An item can by selected by two ways:

1. by its index within the dataModel - the page is scrolled to the page containing the selected item
2. by its index within the visible items (0 up to (itemsPerPage-1))

When the item is added to dataModel, the page is not scrolled and the visible index of the selected item remains the same, unless there are less items in total than itemsPerPage, then a selected item is moved one lower to fill the first page with the new item.

When deleting currently selected item, the next item, if existing, is selected, otherwise the previous item is selected.

Since the data model is from outside, this component have to be notified upon adding/removing items in data model via corresponding handlers itemAdded() and itemDeleted(dataIndex). The adding and deleting of items can be "discovered" in onCountChange handler, but to be able to handle (update the view) deletion of not-selected items (or items ouf of the visible range), the data index of deleted item has to provided. The itemAdded() handler is added for consistency (not needed).

B.3.26 FilterableList

B.3.27 FullScreenThrobber

B.3.28 HorizontalScrollbar

B.3.29 IconButton

B.3.30 InstallWizardOverviewItem

An overview item consists of two parts: the always visibile "big" block, and a smaller secondary block. Both blocks can have an icon and an optional button.

B.3.31 KeyButton

B.3.32 MenuBarButton

B.3.33 MenuItem
A menuitem is a clickable element that is ususally displayed in the menu app screen. Clicking it usually opens a full screen app. The clicked signal is fired when the menuitem is touched.

B.3.34 NumberBullet

B.3.35 NumberSpinner

B.3.36 NumericKeyboard

B.3.37 OnOffToggle

B.3.38 OptionToggle

B.3.39 ParentalControlOverlay

B.3.40 Popup
The base class for any Popup.

B.3.41 ProgressBar

B.3.42 RadioButtonList
In a group of radio buttons, only one radio button can be checked at a time. If the user selects another button, the previously selected button is switched off.

B.3.43 RichTextButton
Component extending StandardButton to support rich text formatting (e.g. html tags) in button text.

B.3.44 RoundedRectangle

B.3.45 Screen
The base class for any screen. Provides default onShow & onHide transitions.

B.3.46 ScrollableSimpleList
Component displaying up to fixed number of items scrollable by up/down buttons. Scrolling is page based and is always scrolled to have the itemsPerPage-th item at the top of the visible items - might be empty lines at the last page when total count is not itemsPerPage multiple.

B.3.47 ScrollBar
Component displaying up to fixed number of items scrollable by up/down buttons. Scrolling is page based.

B.3.48 SelectorWizard

B.3.49 SidePanelButton

B.3.50 SimpleList

B.3.51 SingleLabel

B.3.52 SlidePanelButton

B.3.53 SmartDeviceList
Component displaying list of smart devices - smokeDetectors, plugs or lamps.

Component is build around two list models, one that contains the data and one that containts the visible items. At the end of the list a dash bordered item is added to "Add" a device.

Scrolling is page based using goToPage() method. To add/remove device use addDevice()/removeDevice methods.

B.3.54 StandardButton

B.3.55 StandardCheckBox
A component that represents a checkbox.

B.3.56 StandardRadioButton
A component that represents a radiobutton.

B.3.57 StatusButton

B.3.58 SystrayIcon

A SystrayIcon is a clickable element that is usually displayed in the systray. Clicking it usually opens a full screen app. The clicked signal is fired when the SystrayIcon is touched. The icons are sorted numerically according to the posIndex property.

B.3.59 ThreeStateButton

ThreeStateButton implements button with 3 defined states: "up", "down" and "disabled".

Clickable area with size of the button is created and filled with background color (backgroundUp property).

Button hold only the image (no text). Original image can be rotated using 'imgRotation' property.

For image placed on the button shadow is created (moving original image [+2; +2] with opacity 20%.

Disabled button has no shadow for the icon and color effect is applied on the original image.

States "up" and "down" are handled via mouse click. Icon and button background is different for the states.

B.3.60 Throbber

There are small dots in a circle and bigger dots, which is used for animation (changing position).

Count of dots is variable (property dotCnt). Every position of dots is precalculated at the begining.

Timer is used for changing position of bigger dots. Timer is running only when component is visible.

B.3.61 Tile

A tile is a clickable element that is ususally displayed in the home screen.

Clicking it usually opens a full screen app. However it can also contain clickable controls.

This type should be extended to create the tiles that are provided by Apps.

The clicked signal is fired when the tile is touched.

B.3.62 TimeNumberSpinner

B.3.63 TimeScale

B.3.64 TipsPopup

B.3.65 Toast

B.3.66 TopTabBar

B.3.67 TopTabButton

B.3.68 TwoStateIconButton

B.3.69 UnFlickable
Container that mimmics the interface of the Flickable container, except it's not flickable.

Usefull to easily switch between flickable and much-lighter non-flickable implementations of the same view.

B.3.70 UnFlickableRectangle
Container that mimmics the interface of the Flickable container, except it's not flickable.

Usefull to easily switch between flickable and much-lighter non-flickable implementations of the same view.

The Rectangle base class allows the container to be colored.

B.3.71 WaitPopup
This popup is used where throbber cannot be used because system is fully loaded and throbber would not be animated.

B.3.72 WarningBox
Static rectangle showing warning message. This warning is meant to be placed on the screen where e.g. uncorrect setting of parameter by user may be dangerous.

B.3.73 WizardFrame