# BRisa Qt Documentation

*Release 0.1.3*

**Arthur de Souza Ribeiro**
**Leandro Melo de Sales**

April 16, 2010

# CONTENTS

Contents:

CHAPTER

ONE

# ABOUT

This document describes the BRisa Project, its goals and what it provides.

## 1.1 BRisa Project

Under development since 2007 at the Laboratory of Embedded Systems and Pervasive computing located at Universidade Federal de Campina Grande - UFCG (Campina Grande - Brazil), the project general goal is to provide UPnP solutions.

The UPnP solution is now BRisa Qt, a UPnP framework written in Qt for deploying devices and control points.

News and updates will always be posted no our front page.

## 1.2 People

Team Leader and Project Manager: Leandro de Melo Sales <*leandroal at gmail dot com*>

**Student Developers:**

- André Dieb Martins <*andre.dieb at gmail dot com*>
- André Luiz de Guimarães <*andre.leite at ee.ufcg.edu.br*>
- Arthur de Souza Ribeiro <*arthurdesribeiro at gmail dot com*>
- Caio Nobrega <*caionobrega0 at gmail dot com*>
- Flavio Fabricio <*flaviofabricio at gmail dot com*>
- Gabriel Bezerra <*gabriel.bezerra at gmail dot com*>
- Renato Souto <*rw.souto at gmail dot com*>

CHAPTER

TWO

# GLOSSARY

This is a basic glossary recommended reading before starting reading the documentation itself.

**BRisa Project** project under development at Laboratory of Embedded Systems and Pervasive Computing, financed by INdT.

**BRisa Qt** a Qt upnp framework developed by the BRisa Project.

**upnp** also known as UPnP, short for universal plug and play, protocol defined by the UPnP Forum. Documents can be found at http://www.upnp.org.

**device** for our purposes, when refering to a device, we will be refering to a UPnP device, that is, an entity capable of joining the UPnP network (thus capable of performing its role on the UPnP Architecture.

**service** for our purposes, a service should be considered an UPnP service, that is, a service provider of actions. For example, a Content Directory service provides a Browse action that enables others to browse through a remote directory on the service location.

**control point** an entity of the UPnP network capable of controlling devices

CHAPTER

THREE

# INSTALLATION — INSTALLATION AND COMPILATION

These are the ways to compile and install BRisa Qt in each platform

## 3.1 linux — Compiling/Installing on Linux

The compilation and installation proccess in Linux is very simple, and don't take too much time.

### 3.1.1 Compiling Brisa Qt in linux

The first thing you need to do is to install libqxt in your system. Brisa Qt tree makes this very easy for you and have it own version of libqxt with changes made to make things work fine in Brisa.

The first thing you need to do is, in the Brisa Tree go to the libqxt directory and do

```
$ ./configure
$ make
$ sudo make install
```

This will install libqxt in your computer, after that it's easy to compile Brisa Qt in windows, you have just to go to the main directory and do

```
$ ./configure
$ make
```

And Brisa Qt will be successfully compiled.

### 3.1.2 Installing Brisa Qt on Linux

After installing libqxt in your system, (see the section Compiling Brisa Qt in linux), to install the Brisa Qt in your system you have just do

```
$ sudo make install
```

Or the installation can be done by using our .debs files, first if you don't have the Brisa's libqxt lib installed, do

```
$ sudo dpkg -i libqxt-dev_0.5.0maemo_all.deb
```

And after that

```
$ sudo dpkg -i libbrisa-dev_0.4.0maemo_all.deb
```

Brisa Qt has some examples, like Control Point and Devices, that can be installed by doing

```
$ sudo dpkg -i BrisaDebExamples_Linux.deb
```

## 3.2 `maemo4` — Compiling/Installing on Maemo 4

The compilation proccess for maemo is done in Scratchbox, because the device have some hardware limitation that can disrupt the proccess, the installation is done by .deb packages.

### 3.2.1 Compiling Brisa Qt in Maemo 4 (Via Scratchbox)

The first thing you need to do is to install libqxt in your system. You can do that by direct installing the .deb file

```
$ dpkg -i libqxt-dev_0.5.0maemo_all.deb
```

Or by compilling in Scratchbox and then install it

```
$ sudo  apt-get install cdbs             (Done in device and Scratchbox)
$ sudo  apt-get install libqt4-dev       (Done in device and Scratchbox)
$ cd /brisa-cpp/trunk/dependencies/libqxt
$ dpkg-buildpackage
$ cd ..
$ dpkg -i libqxt-dev_0.5.0maemo_all.deb  (Done in device and Scratchbox)
```

And after that too compile Brisa Qt you have to do(via Scratchbox)

```
$ cd /brisa-cpp/trunk/brisa-cpp
$ dpkg-buildpackage
```

And you're going to be ready to install the .deb file

### 3.2.2 Installing Brisa Qt on Maemo 4

To install Brisa Qt in Maemo 4 the main thing you gotta do is to install from the .deb file by doing

```
$ sudo dpkg -i libbrisa-dev_0.4.0maemo_all.deb
```

Remember that you need to have libqxt installed (look the Compiling Brisa Qt in Maemo 4 (Via Scratchbox) session).

## 3.3 `maemo5` — Compiling/Installing on Maemo 5

The compilation proccess for maemo is done in Scratchbox, because the device have some hardware limitation that can disrupt the proccess, the installation is done by .deb packages. The proccess is VERY similar to Maemo 4 one

### 3.3.1 Compiling Brisa Qt in Maemo 5 (Via Scratchbox)

The first thing you need to do is to install libqxt in your system. You can do that by direct installing the .deb file

```
$ sudo dpkg -i libqxt-dev_0.5.0maemo_all.deb
```

Or by compilling in Scratchbox and then install it

```
$ sudo  apt-get install cdbs             (Done in device and Scratchbox)
$ sudo  apt-get install libqt4-dev       (Done in device and Scratchbox)
$ cd /brisa-cpp/trunk/dependencies/libqxt
$ dpkg-buildpackage
$ cd ..
$ dpkg -i libqxt-dev_0.5.0maemo_all.deb  (Done in device and Scratchbox)
```

And after that too compile Brisa Qt you have to do(via Scratchbox)

```
$ cd /brisa-cpp/trunk/brisa-cpp
$ dpkg-buildpackage
```

And you're going to be ready to install the .deb file

### 3.3.2 Installing Brisa Qt on Maemo 5

To install Brisa Qt in Maemo 5 the main thing you gotta do is to install from the .deb file by doing

```
$ sudo dpkg -i libbrisa-dev_0.4.0maemo_all.deb
```

Remember that you need to have libqxt installed (look the Compiling Brisa Qt in Maemo 5 (Via Scratchbox) session).

The proccess is equal to maemo 4 one, but pay attention when making applications it's not the same thing.

## 3.4 `windows` — Compiling/Installing on Windows

The compilation and installation proccess in Windows is very simple too, and don't take too much time.

### 3.4.1 Compiling Brisa Qt in Windows

The first thing you need to do is to install libqxt in your system. Just like in linux, Brisa Qt tree makes this very easy for you and have it own version of libqxt with changes made to make things work fine in Brisa.

The first thing you need to do is, in the Brisa Tree go to the libqxt directory and do

```
$ configure.bat
$ mingw32-make
$ mingw32-make install
```

This will install libqxt in your computer, after that it's easy to compile Brisa Qt in windows, you have just to go to the main directory and do

```
$ configure.bat
$ mingw32-make
```

And Brisa Qt will be successfully compiled.

### 3.4.2 Installing Brisa Qt on Linux

After installing libqxt in your system, (see the section Compiling Brisa Qt in Windows), to install the Brisa Qt in your system you have just do

```
$ mingw32-make install
```

And Brisa Qt is finally installed in your computer.

CHAPTER

FOUR

# CORE-UTILS — FRAMEWORK'S CORE AND UTILS

These are the core and utils namespaces of Brisa Qt. Each section describes a class and how to use it.

## 4.1 config — Configuration facilities

Just like in *python-brisa* Brisa Qt also provides a very simple configuration API.

### 4.1.1 Using the Built-In Configuration Manager

```
QString configPath("./");

//generate my own config
QHash<QString,QString> state;
state["brisaPython.owner"] = "owner1";
state["brisaPython.version"] = "0.10.0";
state["brisaPython.encoding"] = "utf-8'";
state["brisaC++.owner"] = "owner2";
state["brisaC++.version"] = "0.1.0";
state["brisaC++.encoding"] = "utf-8";

BrisaConfigurationManager *myConfig = new BrisaConfigurationManager(configPath, state);

//save the config
myConfig->save();
qDebug() << "value: " << myConfig->getParameter("brisaC++","owner");
myConfig->setParameter("brisaC++", "owner", "newOwner");
myConfig->save();

//update the config
myConfig->update();
qDebug() << "value: " << myConfig->getParameter("brisaC++","owner");
```

You can create Configuration Managers by using the BrisaConfigurationManager class, and from it you can set parameter and get parameters in an easy way. The configuration is saved and can be used later.

## 4.2 `log` — Logging facilities

Brisa Qt provides logging functions with a colored logging feature.

By the log function, it's possible to create a debug, warning, critical or fatal messages the proccess is simple and easy to understand

### 4.2.1 Global Logger

This module provides a global (or root) logger, which can be used simply by

```
#include <BrisaUtils/BrisaLog>

brisaLogInitialize()

qDebug() << "My debug message";
qWarning() << "My debug message";
qCritical() << "My debug message";
qFatal() << "My debug message";
```

## 4.3 `network` — Network facilities

Brisa Qt has some functions which will simplify networking related tasks.

### 4.3.1 Using the getIp(interface) function

This little example shows a simple and generic use of getIp(interface) function

```
#include <BrisaUtils/BrisaNetwork>

QString ip = getIp("wlan0");
```

It will return a QString containing the ip in the passed interface

### 4.3.2 Getting port

To get the port is very simple too

```
#include <BrisaUtils/BrisaNetwork>

quint16 port = getPort();
```

This will return a quint16 containing a port number.

## 4.4 `webserver` — Webserver facilities

### 4.4.1 How to create a simple webservice using BRisa Qt.

**BRisa uses the libqxt to implement its webservice module, it also has other great extensions besides web modules. To create a si**

- BrisaWebServer
- BrisaWebServiceProvider
- BrisaWebService

We start our webserver creating a BrisaWebServer using new, passing an Ip address and port number.

```
BrisaWebServer *webserver = new BrisaWebserver(QHostAddress(ipAddress), port);
```

After that we need something to manage our services, thats where the BrisaWebServiceProvider comes in, we build it passing the webserver we have just created. After that we call the "addService" method from webserver to add the service manager to the root of the webserver, passing the desired url path. A BrisaWebServiceProvider itself is a webservice. It just doesnt emits the signals we want.

```
BrisaWebServiceProvider *webserviceManager = new BrisaWebServiceProvider(webserver, this);
webserver->addService("hello",webserviceManager);
```

We now can create the webservice(or webservices) passing our webserver to tie everything up. After that we call the "addService" method from our webserviceManager object we created, passing the url path we want for our webservice. So our service will be in the path "IP:PORT/hello/world".

```
BrisaWebService *hello = new BrisaWebService(webserver, this);
webserviceManager->addService("world", hello);
```

When a request arrives at our webservice two signals are emmited, they are:

```
void genericRequestReceived(const QString &method,
                            const QMultiHash<QString, QString> &headers,
                            const QByteArray &requestContent,
                            int sessionId,
                            int requestId);

void genericRequestReceived(BrisaWebService *service,
                            QMultiHash<QString, QString>,
                            QString requestContent);
```

So, simply connect one(or both) of the signals to a slot and treat the request the way you want to. If you want to answear the request, simply call one of the "respond()" methods from BrisaWebService. Thats all.

CHAPTER

FIVE

# UPNP — UPNP MODULES

These are the UPnP classes of Brisa Qt. Each section describes a module and its usage.

## 5.1 device — Device building and deploying

These are the UPnP modules of *python-brisa*. Each section describes a module and its usage.

### 5.1.1 device — Device classes

Device-side device class used for implementing and deploying UPnP devices.

#### Device class

Class that represents a device.

#### Attributes

#### Methods

### 5.1.2 service — Service classes

#### Implementing a simple service.

There is two ways of implement a service: using your own scpd.xml file or programatically describes it.

#### Implementing a service with a scpd.xml file.

Write your scpd.xml file.

```
<?xml version="1.0" encoding="utf-8"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
    <specVersion>
        <major>1</major>
        <minor>0</minor>
    </specVersion>
    <actionList>
        <action>
            <name>MyMethod</name>
            <argumentList>
                <argument>
                    <name>TextIn</name>
                    <direction>in</direction>
                    <relatedStateVariable>A_ARG_TYPE_Textin</relatedStateVariable>
                </argument>
                <argument>
                    <name>TextOut</name>
                    <direction>out</direction>
                    <relatedStateVariable>A_ARG_TYPE_Textout</relatedStateVariable>
                </argument>
            </argumentList>
        </action>
    </actionList>
    <serviceStateTable>
        <stateVariable sendEvents="no">
            <name>A_ARG_TYPE_Textout</name>
            <dataType>string</dataType>
        </stateVariable>
        <stateVariable sendEvents="yes">
            <name>A_ARG_TYPE_Textin</name>
            <dataType>string</dataType>
        </stateVariable>
    </serviceStateTable>
</scpd>
```

Now, create your service class and inherits from BrisaService class. You will need to specify the service name, the service type, the scpd.xml file and implement a Brisa Action to run the service action. The BrisaAction must have the same name as in the XML

First, define the constant variables in the beginning of the code.

```
#define SERVICE_TYPE "urn:schemas-upnp-org:service:MyService:1"
#define SERVICE_ID "MyService"
```

After this, let's include the Brisa libraries that are used to create the Actions and the services.

```
#include <BrisaUpnp/BrisaAction>
#include <BrisaUpnp/BrisaService>

using namespace BrisaUpnp;
```

Now, create the actions that are going to be used by the service and after that finish the service implementation. The run method is virtual and have to be implemented to do the action perform.

```
class MyMethod : public BrisaAction
{
    public:
        MyMethod(BrisaService *service) : BrisaAction("MyMethod", service) {}

    private:
        QMap<QString, QString> run(const QMap<QString, QString> &inArguments)
        {
            QMap<QString, QString> outArgs;
            QString inArg = inArguments["TextIn"];
            outArgs.insert("TextOut", inArg + "Out!!");
            return outArgs;
        }
};
```

Now, we have the actions created, so let's start creating the service to finish the service part.

```
class MyService : public BrisaService
{
    public:
        MyService() : BrisaService(SERVICE_TYPE,
                                     SERVICE_ID)
        {
            addAction(new MyMethod(this));
        }
};
```

Now, when implementing a device we should do

```
MyService *myService = new MyService();
myService->setDescriptionFile("myservice-scpd.xml.xml");
```

And when the service is added to a device and we call start in device, all the attributes of the service are going to be initialized.

### Implementing a service without a scpd.xml file.

You will need to specify your service definition programatically. Don't forget to set the service's parameters and to create the action that is going to belong to the service.

```
#include <BrisaUpnp/BrisaAction>
#include <BrisaUpnp/BrisaService>

using namespace BrisaUpnp;

#define SERVICE_TYPE "urn:schemas-upnp-org:service:MyService:1"

class MyMethod : public BrisaAction
{
    public:
        MyMethod(BrisaService *service) : BrisaAction("MyMethod", service) {}

    private:
        QMap<QString, QString> run(const QMap<QString, QString> &inArguments)
        {
            QMap<QString, QString> outArgs;
            QString inArg = inArguments["TextIn"];
            outArgs.insert("TextOut", inArg + "Out!!");
            return outArgs;
        }
};

class MyService : public BrisaService
{
    public:
        MyService(const QString &serviceType = SERVICE_TYPE, const QString &serviceId = "",
                  const QString &scpdUrl = "", const QString &controlUrl = "",
                  const QString &eventSubUrl = "", const QString &host = "",
                  QObject *parent = 0);
};
```

In the source file we do

```
#include "myservice.h"

MyService::MyService(const QString &serviceType, const QString &serviceId,
                     const QString &scpdUrl, const QString &controlUrl,
                     const QString &eventSubUrl, const QString &host,
                     QObject *parent) : BrisaService(serviceType, serviceId,
                                                     scpdUrl, controlUrl,
                                                     eventSubUrl, host,
                                                     parent)
{
    this->addStateVariable("Yes", "A_ARG_TYPE_Textin", "string", "", "", "", "");
    this->addStateVariable("Yes", "A_ARG_TYPE_Textout", "string", "", "", "", "");

    MyMethod *myMethod = new MyMethod(this);
    myMethod->addArgument("TextIn", "in", "A_ARG_TYPE_Textin");
    myMethod->addArgument("TextOut", "out", "A_ARG_TYPE_Textout");

    addAction(myMethod);
}
```

## 5.2 `control_point` — Control Point API

These are the UPnP modules of *python-brisa*. Each section describes a module and its usage.

### 5.2.1 `service` — Service classes

#### Subscribe for unicast eventing.

It's very simple to subscribe for service unicast eventing. You need to select the service object that you want to subscribe get the event proxy, and call subscribe method.

To subscribe to a service is simpes using Brisa Qt, and you onyl have to pass the subscription timeout, after that you can renew your can renew your subscription or just do nothing

```
class ControlPoint : BrisaControlPoint

public:
    ControlPoint() : BrisaControlPoint() {};

    void subscribe(BrisaControlPointService *service, int timeout = -1);

    void unsubscribe(BrisaControlPointService *service);

public slots:
    eventReceived(BrisaEventProxy *eventProxy,QMap<QString, QString> map));
```

After that do the method/slot implementation in the source file

```
void ControlPoint::subscribeService(BrisaControlPointService *service, int timeout)
{
    BrisaEventProxy *subscription = this->getSubscriptionProxy(service);

    connect(subscription,SIGNAL(eventNotification(BrisaEventProxy *,QMap<QString, QString>)),
            this, SLOT(eventReceived(BrisaEventProxy *,QMap<QString, QString>)));

    subscription->subscribe(timeout);
}

void ControlPoint::eventReceived(BrisaEventProxy *eventProxy,QMap<QString, QString> map)
{
    Q_UNUSED(subscription);

    qDebug() << "Event Message!";
    for(int i = 0; i < eventVariables.keys().size(); i++) {
        qDebug() << "State Variable: " << eventVariables.keys()[i];
        qDebug() << "Value: " << eventVariables[eventVariables.keys()[i]];
    }
}
```

#### Unsubscribe for unicast eventing.

In order to unsubscribe for service unicast eventing, you need to select the service object that you want to unsubscribe get the event Proxy and call the unsubscribe method

```
void ControlPoint unsubscribe(BrisaService *service)
{
    BrisaEventProxy *unsubscription = this->getSubscriptionProxy(service);

    unsubscription->unsubscribe();
}
```

## 5.3 `SSDP` — SSDP API

These are the classes to create SSDP server and SSDP clients, used in device's and control point respectively

### 5.3.1 `ssdpserver` — SSDP Server implementation

Brisa Qt provides SSDP server (Simple Service Discovery Protocol) class which can be used on the device's side for annoucing the device, its embedded devices and all services..

#### BrisaSSDPServer class

Implementation of a SSDP Server.

### Attributes

The attribute of the BrisaSSDPServer class are:

- running - Bool to tell if server is running or not
- SSDP_ADDR - SSDP address
- SSDP_PORT - SSDP port
- S_SSDP_PORT - QString port, used in bind
- udpSocket - Udp Socket to join multicast group

### Methods and Slots

**isRunning**()
    Returns True if the BrisaSSDPServer is running, False otherwise.

**start**()
    Call this method to join the multicast group and start listening for UPnP msearch responses.

**stop**()
    Sends bye bye notifications and stops the BrisaSSDPServer.

**doNotify(const QString &usn, const QString &location, const QString &st,**()
**const QString &server, const QString &cacheControl)**()
    Sends a UPnP notify alive message to the multicast group with the given information.

**doByeBye**(*const QString &usn, const QString &st*)
    Sends a UPnP notify byebye message to the multicast group with the given information.

**datagramReceived**()
    This slot is called when the readyRead() signal is emmited by the QUdpSocket listening to incoming messages.

**msearchReceived(QHttpRequestHeader *datagram, QHostAddress *senderIp,**()
**quint16 senderPort)**()
    Emits msearchRequestReceived if the incoming message is a valid msearch.

**respondMSearch(const QString &senderIp, quint16 senderPort,**()
**const QString &cacheControl, const QString &date,**()
**const QString &location, const QString &server,**()
**const QString &st, const QString &usn)**()
    Connect this slot to a proper signal to get synchronous response for msearch requests.

## 5.3.2 `ssdpclient` — SSDP Client implementation

Brisa Qt provides SSDP client (Simple Service Discovery Protocol) class which can be used on the control point's side for receiving notification messages

### BrisaSSDPClient class

Implementation of a SSDP Client.

### Attributes

The attribute of the BrisaSSDPClient class are:

- running - Bool to tell if server is running or not
- SSDP_ADDR - SSDP address
- SSDP_PORT - SSDP port
- S_SSDP_PORT - QString port, used in bind
- udpListener - Socket to join multicast group and listen to msearch notification

### Methods and Slots

**`isRunning`**`()`
  Returns True if the BrisaSSDPClient is running, False otherwise.

**`start`**`()`
  Connects to the MultiCast group and starts the client.

**`stop`**`()`
  Stops the client.

**`datagramReceived`**`()`
  Receives UDP datagrams from a QUdpSocket.

**`notifyReceived`**`(`*QHttpRequestHeader *datagram*`)`
  Parses the UDP datagram received from “datagramReceived()”.

CHAPTER

SIX

# CODE EXAMPLES

These are some examples to show how simple it is to develop using BRisa.

## 6.1 Creating a Binary Light Device

This is the implementation of a Binary Light Device, which the specifications you can find here.

Before you create a Device, it's very important that you create first the service, that the device is going to use.

First, define the constant variables in the beginning of the code.

```
#define SERVICE_TYPE "urn:schemas-upnp-org:service:SwitchPower:1"
#define SERVICE_ID "SwitchPower"
#define SERVICE_XML_PATH "/SwitchPower/SwitchPower-scpd.xml"
#define SERVICE_CONTROL "/SwitchPower/control"
#define SERVICE_EVENT_SUB "/SwitchPower/eventSub"
```

After this, let's include the Brisa libraries that are used to create the Actions and the services.

```
#include <BrisaUpnp/BrisaAction>
#include <BrisaUpnp/BrisaService>

using namespace BrisaUpnp;
```

Now, create the actions that are going to be used by the service and after that finish the service implementation. The run method is virtual and have to be implemented for the action perform.

```
class GetStatus : public BrisaAction
{
    public:
        GetStatus(BrisaService *service) : BrisaAction("GetStatus", service) {}

    private:
        QMap<QString, QString> run(const QMap<QString, QString> &inArguments)
        {
            Q_UNUSED(inArguments)

            QMap<QString, QString> outArgs;
            outArgs.insert("ResultStatus", this->getStateVariable("Status")
                                               ->getAttribute(BrisaStateVariable::Value));
            return outArgs;
        }
};

class GetTarget : public BrisaAction
{
    public:
        GetTarget(BrisaService *service) : BrisaAction("GetTarget", service) {}

    private:
        QMap<QString, QString> run(const QMap<QString, QString> &inArguments)
        {
            Q_UNUSED(inArguments)

            QMap<QString, QString> outArgs;
            outArgs.insert("RetTargetValue", this->getStateVariable("Target")
                                        ->getAttribute(BrisaStateVariable::Value));
            return outArgs;
        }
};

class SetTarget : public BrisaAction
{
    public:
        SetTarget(BrisaService *service) : BrisaAction("SetTarget", service) {}

    private:
        QMap<QString, QString> run(const QMap<QString, QString> &inArguments)
        {
            this->getStateVariable("Target")->setAttribute(BrisaStateVariable::Value,
                                                           inArguments["NewTargetValue"]);
            this->getStateVariable("Status")->setAttribute(BrisaStateVariable::Value,
                                                           inArguments["NewTargetValue"]);

            QMap<QString, QString> outArgs;
            return outArgs;
        }
};
```

Now, we have the actions created, so let's start creating the service to finish the service part.

```
class SwitchPower : public BrisaService
{
    public:
        SwitchPower() : BrisaService(SERVICE_TYPE,
                                     SERVICE_ID,
                                     SERVICE_XML_PATH,
                                     SERVICE_CONTROL,
                                     SERVICE_EVENT_SUB)
        {
            addAction(new SetTarget(this));
            addAction(new GetTarget(this));
            addAction(new GetStatus(this));
        }
};
```

Note that in the constructor we have put the actions inside the service, so that they are callable now. Now, we start implementing the Device, the first thing you have to do is to include Brisa libs, the service file and declare the namespace

```
#include <BrisaUpnp/BrisaDevice>
#include "switchPower.h"

using namespace BrisaUpnp;
```

Now, we define the device's constants

```
#define DEVICE_TYPE                 "urn:schemas-upnp-org:device:BinaryLight:1"
#define DEVICE_FRIENDLY_NAME        "Binary Light Device"
#define DEVICE_MANUFACTURER         "Brisa Team. Embedded Laboratory and INdT Brazil"
#define DEVICE_MANUFACTURER_URL     "https://garage.maemo.org/projects/brisa"
#define DEVICE_MODEL_DESCRIPTION    "An UPnP Binary Light Device"
#define DEVICE_MODEL_NAME           "Binary Light Device"
#define DEVICE_MODEL_NUMBER         "1.0"
#define DEVICE_MODEL_URL            "https://garage.maemo.org/projects/brisa"
#define DEVICE_SERIAL_NUMBER        "1.0"
```

And after all these things we can start the device's implementation by setting the slots that are going to be used, the attributes and the methods. Note that the device is going to have two states variables and will have a BrisaDevice as an attribute.

```
class Device : public QWidget
{
    Q_OBJECT

public:
    Device(QWidget *parent = 0);
    ~Device();

public slots:
    void statechanged(BrisaStateVariable *);

private:
    BrisaDevice binaryLight;
    BrisaStateVariable *status;
    BrisaStateVariable *target;

    //METHODS FOR UDN CREATION
    QString createUdn();
    int get1RandomNumber();
    int get2RandomNumber();
    int get3RandomNumber();
};
```

Now, our header file is ready and we have to start the implementation of the device, first let's do the constructor

```
Widget::Widget(QWidget *parent) :
    QWidget(parent),
    binaryLight(DEVICE_TYPE,
                DEVICE_FRIENDLY_NAME,
                DEVICE_MANUFACTURER,
                DEVICE_MANUFACTURER_URL,
                DEVICE_MODEL_DESCRIPTION,
                DEVICE_MODEL_NAME,
                DEVICE_MODEL_NUMBER,
                DEVICE_MODEL_URL,
                DEVICE_SERIAL_NUMBER,
                createUdn())
{
    SwitchPower *switchPower = new SwitchPower();
    switchPower->setDescriptionFile("SwitchPower-scpd.xml");
    binaryLight.addService(switchPower);
    binaryLight.start();

    this->status = binaryLight.getServiceByType("urn:schemas-upnp-org:service:SwitchPower:1")->getVar
    this->target = binaryLight.getServiceByType("urn:schemas-upnp-org:service:SwitchPower:1")->getVar

    connect(status, SIGNAL(changed(BrisaStateVariable *)), this, SLOT(statechanged(BrisaStateVariable
}
```

Note that we initialized our Device attribute and added the service that was set by the xml description file to it, the state variables we've initialized by catching from the service, so, now it's possible now to see if a value of a variable changes.

Here is what we do to create the device's UDN

```
QString Widget::createUdn()
{
    QString randNumber1;
    QString randNumber2;
    QString randNumber3;
    QString randNumber4;
    QString randNumber5;
    QString randNumber6;
    QString randNumber7;
    QString udn;

    randNumber1.setNum(get3RandomNumber());
    randNumber2.setNum(get1RandomNumber());
    randNumber3.setNum(get1RandomNumber());
    randNumber4.setNum(get2RandomNumber());
    randNumber5.setNum(get1RandomNumber());
    randNumber6.setNum(get1RandomNumber());
    randNumber7.setNum(get3RandomNumber());

    udn.append("uuid:");
    udn.append(randNumber1);
    udn.append("a");
    udn.append(randNumber2);
    udn.append("fa");
    udn.append(randNumber3);
    udn.append("-cf");
    udn.append(randNumber4);
    udn.append("-");
    udn.append(randNumber5);
    udn.append("cc");
    udn.append(randNumber6);
    udn.append("-");
    udn.append(randNumber7);
    udn.append("d-");
    udn.append("0af4c615cecb");

    return udn;
}
```

```
int Widget::get1RandomNumber()
{
    srand(time(NULL));
    return rand() % 10;
}

int Widget::get2RandomNumber()
{
    srand(time(NULL));
    return rand() % 90 + 10;
}

int Widget::get3RandomNumber()
{
    srand(time(NULL));
    return rand() % 900 + 100;
}
```

After these steps we create the Slot implementation to show when the variable changes.

```
void Widget::statechanged(BrisaStateVariable *var)
{
    if(var->getAttribute(BrisaStateVariable::Value) == "1") {
        qDebug() << "Light Switched on";
    } else {
        qDebug() << "Light Switched off";
    }
}
```

And the last thing we've gotten to do is the main that will initialize the device.

```
#include <QtGui/QApplication>
#include "light.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget window;

    return a.exec();
}
```

Remeber to change your .pro file and put.

```
CONFIG += BRISA
BRISA += upnp core utils
```

Thats all it takes to implement the Binary Light Device.

You could follow these steps to implement any service/device, but remember that for now you'll have to write the service's xml yourself.

You can downlaod easily each file click on the name to download the file **light.h**, **light.cpp**, **switchPower.h**, **main.cpp**, **BrisaBinaryLight.pro**

and the service's xml **Here**.

## 6.2 Creating a simple Control Point

This is an example of a simple Control Point, in this example we will implement a Binary Light command line Control Point, but this may help you to create any Control Point. This Control Point has the following features:

- Searching for devices
- Listing devices
- Handling events (new device located, removed device)

As we sad above, our ControlPoint is a command line one, and we'll have to implement two things, the ControlPoint that will hold the devices and the thread to receive the commands. The first thing to do is to do the right includes in the code.

```
#include <BrisaUpnp/BrisaControlPoint> // To create the Control Point
#include <QList>    // To store the devices
#include <QString>
#include <QtDebug>  // For debugging purposes
#include <QTextStream>  // For receiving the commands
#include <QCoreApplication>
#include <iostream>
```

As we sad, our control point is to interact with Binary Light devices, so we define some constants to use in the code.

```
#define BINARY_LIGHT_TYPE    "urn:schemas-upnp-org:device:BinaryLight:1"  //The binary light type
#define SERVICE_SWITCH_POWER "urn:schemas-upnp-org:service:SwitchPower:1" //Service that contains the
```

After doing this, let's create and implement the ControlPoint class, we decided to inherit from BrisaControlPoint, but you could have a BrisaControlPoint object to work as a ControlPoint too.

Our Control Point class is going to show when devices enter or leave the network, is going to list device to the user, and is going to interact to the device too, by calling some actions from them, for this we'll have to have the following class implementation with the attributes, a few methods and the main slots( yes, slots, because the commands are going to come from the Thread)

```
class ControlPoint : BrisaControlPoint
{
    Q_OBJECT
    public:
        ControlPoint();   //ControlPoint Constructor
        ~ControlPoint();  //ControlPoint Destructor


    private:
        QString listServices(BrisaControlPointDevice *dev);
        QString listEmbeddedDevices(BrisaControlPointDevice *dev);

        QList <BrisaControlPointDevice*> devices;  //List to store the devices
        BrisaControlPointDevice *selected;         //Device to interact
        HandleCmds *handle;                        //Thread that handle the commands

    private slots:
        void help();                  // Lists all commands
        void setLight(int index);     // Selects a device
        void getTarget();             // Calls the getTarget action
        void getStatus();             // Calls the getStatus action
        void turnOn();                // Turns on the light
        void turnOff();              // Turns off the light
        void exit();                 // Exits the application
        void list();                 // Lists all the devices;
        void setTargetResponse(QString response, QString method); // Gets the SetTarget response
        void getTargetResponse(QString response, QString method); // Gets the GetTarget response
        void getStatusResponse(QString response, QString method); // Gets the GetStatus response
        void onNewDevice(BrisaControlPointDevice *dev);    //Slot for when a device comes in the netw
        void onRemovedDevice(QString desc);                //Slot for when a device leaves the networ
};
```

Now, we finished the Control Point implementation in the header file, so let's implement all methods and slots that were declared, the first important thing to do is to implement the constructor method that is going to have all the connects and will start the Device's discovery and the commands Thread.

```
#include "controlpoint.h"

ControlPoint::ControlPoint() : BrisaControlPoint()
{
    this->selected = NULL;
    handle = new HandleCmds();

    connect(this, SIGNAL(deviceFound(BrisaControlPointDevice*)), this, SLOT(onNewDevice(BrisaControlP
    connect(this, SIGNAL(deviceGone(QString)), this, SLOT(onRemovedDevice(QString)), Qt::DirectConnec

    this->start();
    this->discover();

    connect(handle, SIGNAL(leave()), this, SLOT(exit()));
    connect(handle, SIGNAL(list()), this, SLOT(list()));
    connect(handle, SIGNAL(help()), this, SLOT(help()));
    connect(handle, SIGNAL(getTarget()), this, SLOT(getTarget()));
    connect(handle, SIGNAL(getStatus()), this, SLOT(getStatus()));
    connect(handle, SIGNAL(setLight(int)), this, SLOT(setLight(int)));
    connect(handle, SIGNAL(turnOn()), this, SLOT(turnOn()));
    connect(handle, SIGNAL(turnOff()), this, SLOT(turnOff()));

    handle->start();
}
```

The destructor is simples and will have the stop command for the discovery.

```
ControlPoint::~ControlPoint()
{
    delete handle;
    this->stop();
}
```

To handle when a device comes or leave the network we show simple messages for the user and store/delete in/from the list.

```
void ControlPoint::onNewDevice(BrisaControlPointDevice *device)
{
    foreach(BrisaControlPointDevice *deviceInside, devices) {
        if(deviceInside->getAttribute(BrisaControlPointDevice::Udn) == device->getAttribute(BrisaCont
            return;
    }
    devices.append(device);
    qDebug() << "Got new device " << device->getAttribute(BrisaControlPointDevice::Udn);
    qDebug() << "Type 'list' to see the whole list";
}

void ControlPoint::onRemovedDevice(QString desc)
{
    foreach(BrisaControlPointDevice *dev, devices) {
        if(dev->getAttribute(BrisaControlPointDevice::Udn) == desc)
            devices.removeAll(dev);
            qDebug() << "Device is gone: " << desc;
    }
}
```

Now the functions that are going to perform the actions by the command handler, and the private functions to show the Services from a device and the Embedded Devices of it

```
void ControlPoint::help()
{
    qDebug() << "Available commands:";
    qDebug() << "exit";
    qDebug() << "help";
    qDebug() << "set_light <dev number>";
    qDebug() << "get_status";
    qDebug() << "get_target";
    qDebug() << "turn <on/off>";
    qDebug() << "stop";
    qDebug() << "list";
}

void ControlPoint::exit()
{
    handle->wait();
    handle->exit();
    QCoreApplication::exit();
}

void ControlPoint::list()
{
    int count = 0;
    foreach(BrisaControlPointDevice *dev, this->devices)
    {
        qDebug() << "Device no: " << count;
        qDebug() << "UDN: " << dev->getAttribute(BrisaControlPointDevice::Udn);
        qDebug() << "Name: " << dev->getAttribute(BrisaControlPointDevice::ModelName);
        qDebug() << "Device Type: " << dev->getAttribute(BrisaControlPointDevice::DeviceType);
        qDebug() << "Services: " << this->listServices(dev);
        qDebug() << "Embedded Devices: " << this->listEmbeddedDevices(dev);
        qDebug() << "";
        count++;
    }
}

QString ControlPoint::listServices(BrisaControlPointDevice *device)
{
    QString services = "";
    QString separator = "";
    foreach(BrisaControlPointService *service, device->getServiceList()) {
        services += separator + service->getAttribute(BrisaControlPointService::ServiceType);
        separator = ", ";
    }
    return "[" + services + "]";
}

QString ControlPoint::listEmbeddedDevices(BrisaControlPointDevice *device)
{
    QString embeddedDevices = "";
    QString separator = "";
    foreach(BrisaControlPointDevice *embedded, device->getEmbeddedDeviceList()) {
        embeddedDevices += separator + embedded->getAttribute(BrisaControlPointDevice::DeviceType);
        separator = ", ";
    }
    return "[" + embeddedDevices + "]";
}

void ControlPoint::setLight(int index)
{
    if(this->devices.size() <= index) {
        qDebug() << "Binary Light Number not found. Please run list and check again.";
        return;
    }
    if(this->devices[index]->getAttribute(BrisaControlPointDevice::DeviceType) != BINARY_LIGHT_TYPE)
        qDebug() << "Please, choose a Binary Light device";
        return;
    }
    this->selected = this->devices[index];
}
```

The action calls function(turn on, turn off, getTarget, getStatus) need a slot to receive the action call response teh call is made in a simple way direct from the BrisaControlPointService Note that in the call we need to pass a QMap, that contains the paramenter name and the value that is going to be passed all these values are strings. If there aren't parameters, so you call with an empty map

```
void ControlPoint::getTarget()
{
    if(this->selected == NULL) {
        qDebug() << "Binary Light Device not select, please run 'set_light <dev_number>'";
        return;
    }
    QMap<QString, QString> param;
    BrisaControlPointService *service = this->selected->getServiceByType(SERVICE_SWITCH_POWER);
    connect(service, SIGNAL(requestFinished(QString, QString)), this, SLOT(getTargetResponse(QString,
    service->call("GetTarget", param);
}

void ControlPoint::getTargetResponse(QString response, QString method)
{
    if(method == "GetTarget") {
        if(response == QString("0")) {
            qDebug() << "Binary Light target is off";
        } else if(response == QString("0")){
            qDebug() << "Binary Light target is on";
        } else {
            qDebug() << response;
        }
    }
```

```
}

void ControlPoint::getStatus()
{
    if(this->selected == NULL) {
        qDebug() << "Binary Light Device not select, please run 'set_light <dev_number>'";
        return;
    }
    QMap<QString, QString> param;
    BrisaControlPointService *service = this->selected->getServiceByType(SERVICE_SWITCH_POWER);
    connect(service, SIGNAL(requestFinished(QString, QString)), this, SLOT(getStatusResponse(QString,
    service->call("GetStatus", param);
}

void ControlPoint::getStatusResponse(QString response, QString method)
{
    if(method == "GetStatus") {
        if(response == QString("0")) {
            qDebug() << "Binary Light status is off";
        } else {
            qDebug() << "Binary Light status is on";
        }
    }
}

void ControlPoint::turnOn()
{
    if(this->selected == NULL) {
        qDebug() << "Binary Light Device not select, please run 'set_light <dev_number>'";
        return;
    }
    QMap<QString, QString> param;
    BrisaControlPointService *service = this->selected->getServiceByType(SERVICE_SWITCH_POWER);
    connect(service, SIGNAL(requestFinished(QString, QString)), this, SLOT(setTargetResponse(QString,
    param["NewTargetValue"] = "1";
    service->call("SetTarget", param);
}

void ControlPoint::turnOff()
{
    if(this->selected == NULL) {
        qDebug() << "Binary Light Device not select, please run 'set_light <dev_number>'";
        return;
    }
    QMap<QString, QString> param;
    BrisaControlPointService *service = this->selected->getServiceByType(SERVICE_SWITCH_POWER);
    connect(service, SIGNAL(requestFinished(QString, QString)), this, SLOT(setTargetResponse(QString,
    param["NewTargetValue"] = "0";
    service->call("SetTarget", param);
}

void ControlPoint::setTargetResponse(QString response, QString method)
{
    if(method == "SetTarget") {
        if(response == QString(""))
            qDebug() << "Turning Binary Light to selected status";
        else
            qDebug() << response;
    }
}
```

The way on how we get the commands(Thread) is not implemented yet, but the implementaion is very simple. On the header file we create the QThread, note that the signals are to be passed to the ControlPoint to perform the actions

```
class HandleCmds : public QThread
{
    Q_OBJECT
    public:
        void run() {
            QTextStream stream(stdin);
            do{
                QString line;
                QCoreApplication::processEvents();
                std::cout << ">>> ";
                line = stream.readLine();
                if(line == "exit") {
                    emit leave();
                    running = false;
                } else if(line == "list") {
                    emit list();
                } else if(line == "help"){
                    emit help();
                } else if(line == "get_target"){
                    emit getTarget();
                } else if(line == "get_status"){
                    emit getStatus();
                } else if(line == "turn on"){
                    emit turnOn();
                } else if(line == "turn off"){
                    emit turnOff();
                } else {
                    if (line.split(" ").size() == 2) {
                        if(line.split(" ")[0] == "set_light") {
                            emit setLight(line.split(" ")[1].toInt());
                        } else {
                            qDebug() << "Wrong usage, try 'help' to see the commands";
                        }
                    } else {
                        qDebug() << "Wrong usage, try 'help' to see the commands";
                    }
                }
            } while(running);
        }

    private:
        void setRunningCmds(bool running) { this->running = running; }

        bool running;

    signals:
        void leave();
        void list();
        void help();
        void setLight(int i);
        void turnOn();
        void turnOff();
        void getTarget();
        void getStatus();
};
```

After that we call it in the Control Point as it was showed

```
HandleCmds *handle = new HandleCmds();

connect(handle, SIGNAL(leave()), this, SLOT(exit()));
connect(handle, SIGNAL(list()), this, SLOT(list()));
connect(handle, SIGNAL(help()), this, SLOT(help()));
connect(handle, SIGNAL(getTarget()), this, SLOT(getTarget()));
connect(handle, SIGNAL(getStatus()), this, SLOT(getStatus()));
connect(handle, SIGNAL(setLight(int)), this, SLOT(setLight(int)));
connect(handle, SIGNAL(turnOn()), this, SLOT(turnOn()));
connect(handle, SIGNAL(turnOff()), this, SLOT(turnOff()));

handle->start();
```

The program is done only missing the main, that is very simple

And that's how we created a control point using Brisa Qt in an easy way.

You can find the code **here**.

CHAPTER

SEVEN

# COPYRIGHT

CHAPTER

EIGHT

# INDICES AND TABLES

# INDEX