

---

# **Qt BRisa Documentation**

***Release 2.0***

**BRisa Team**

January 05, 2011

---

# Contents

---

<b>1</b>	<b>Glossary</b>	<b>1</b>
<b>2</b>	<b>About</b>	<b>2</b>
2.1	BRisa Project . . . . .	2
2.2	People . . . . .	2
<b>3</b>	<b>installation — Installation and Compilation</b>	<b>3</b>
3.1	linux — Compiling/Installing on Linux . . . . .	3
3.2	maemo4 — Compiling/Installing on Maemo 4 . . . . .	3
3.3	maemo5 — Compiling/Installing on Maemo 5 . . . . .	4
3.4	windows — Compiling/Installing on Windows . . . . .	5
<b>4</b>	<b>core-utils — Framework’s core and utils</b>	<b>6</b>
4.1	config — Configuration facilities . . . . .	6
4.2	log — Logging facilities . . . . .	7
4.3	network — Network facilities . . . . .	7
4.4	webserver — Webserver facilities . . . . .	8
<b>5</b>	<b>upnp — UPnP modules</b>	<b>10</b>
5.1	device — Device building and deploying . . . . .	10
5.2	control_point — Control Point API . . . . .	14
5.3	SSDP — Simple Service Discovery Protocol API . . . . .	15
<b>6</b>	<b>Code Examples</b>	<b>18</b>
6.1	Creating a Binary Light Device . . . . .	18
6.2	Creating a simple Control Point . . . . .	21
<b>7</b>	<b>Wizard — Creating a BRisa Project</b>	<b>29</b>
7.1	About Qt BRisa Wizard Application . . . . .	29
7.2	How crete a Brisa Project . . . . .	30
<b>8</b>	<b>Copyright</b>	<b>36</b>
<b>9</b>	<b>Indices and tables</b>	<b>37</b>



---

# Glossary

---

This is a basic glossary defining the most common words that will appear in this documentation. It is recommend reading it first.

**BRisa Project** BRisa is a project focused on the development of UPnP technologies. It provides an API to building UPnP devices, services and control points. BRisa project is released in MIT License (Python version) and in LGPL License (Qt version).

**Qt BRisa** Brisa project version developed at [Laboratory of Embedded Systems and Pervasive Computing](#), financed by [INdT](#). UPnP framework for Qt.

**UPnP** Short for Universal Plug and Play, a protocol defined by the UPnP Forum. The UPnP architecture offers pervasive peer-to-peer network connectivity of PCs of all form, intelligent appliances, and wireless devices. The UPnP architecture is a distributed, open networking architecture that leverages TCP/IP and the Web to enable seamless proximity networking in addition to control and data transfer among networked devices in the home, office, and everywhere in between. Documents can be found at <http://www.upnp.org>.

**device** For our purposes, when referring to a device, we will be referring to a UPnP device. This is one of the key concepts of the [UPnP Architecture](#) and it's used to make reference to some machine or gadget in the network capable of providing services and advertising its presence.

**service** For our purposes, a service should be considered an UPnP service, in other words, it is a logical functional unit capable of exposing actions and modeling a device state through the use of state variables. For example, a Door device could have a service called ControlDoor which would expose Door state (Open or Closed) and UPnP actions responsible for opening and closing the Door.

**control point** An entity of the UPnP network capable of controlling devices by retrieving device and services descriptions, sending actions to services, etc.

---

# About

---

This document will describe BRisa project objective, how it makes the use of UPnP protocol easier and will cover the basics about using BRisa to create new devices, control points and services.

## 2.1 BRisa Project

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). Qt BRisa is part of BRisa project whose objective is to facilitate the use of UPnP protocol to connect devices.

Qt BRisa does that by providing an API written in Qt for easier creation of devices and control points, as well as actions and services.

News and updates will be posted on BRisa project [front page](#).

## 2.2 People

### Project Managers:

- Leandro de Melo Sales <[leandroal@gmail.com](mailto:leandroal@gmail.com)>
- Thiago Sales <[thiagobrunoms@gmail.com](mailto:thiagobrunoms@gmail.com)>

### Student Developers:

- Camilo Campos <[zeromaisum@gmail.com](mailto:zeromaisum@gmail.com)>
- Danilo Freitas <[dsurviver@gmail.com](mailto:dsurviver@gmail.com)>
- Jeysibel Sousa Dantas <[jeysibel@gmail.com](mailto:jeysibel@gmail.com)>
- Nicholas Alexander <[nickmaclewy222@gmail.com](mailto:nickmaclewy222@gmail.com)>
- Vinícius dos Santos Oliveira <[vini.ipsmaker@gmail.com](mailto:vini.ipsmaker@gmail.com)>
- Willian Silva <[willian.victors@gmail.com](mailto:willian.victors@gmail.com)>

---

# installation — Installation and Compilation

---

This section explains how to compile and install Qt BRisa in the supported platforms.

## 3.1 linux — Compiling/Installing on Linux

The compilation and installation process on Linux is very simple, and doesn't take much time.

### 3.1.1 Installing libqxt on linux

Download latest libqxt version from <http://www.libqxt.org/> and follow install instructions found at [http://dev.libqxt.org/libqxt/wiki/user\\_guide](http://dev.libqxt.org/libqxt/wiki/user_guide)

### 3.1.2 Installing Qt BRisa on Linux

With libqxt installed in your system you just need to compile BRisa project, by going to the main directory (qt-brisa) inside Qt BRisa folder tree and typing

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

And installing it typing

```
$ sudo make install
```

You can also install Qt BRisa using the .deb file running the following command inside the folder you downloaded it

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

## 3.2 maemo4 — Compiling/Installing on Maemo 4

The compilation process for maemo is done using Scratchbox, because of hardware limitation that can disrupt the process, the installation is done by .deb packages.

### 3.2.1 Compiling Qt BRisa 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 typing the following command inside the folder you downloaded the file

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

Or by compiling in Scratchbox and then installing it

```
$ sudo apt-get install cdbx (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 you need to compile Qt BRisa (via Scratchbox) typing

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

Now you are ready to install Qt BRisa

### 3.2.2 Installing Qt BRisa on Maemo 4

To install Qt BRisa in Maemo 4 you just need to download the libbrisa-dev\_0.4.0maemo\_all.deb file and type the following command inside the folder you put it

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

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

## 3.3 maemo5 — Compiling/Installing on Maemo 5

The compilation process for maemo is done using Scratchbox, because of hardware limitation that can disrupt the process, the installation is done by .deb packages. The process is similar with the Maemo4 one.

### 3.3.1 Compiling Qt BRisa 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 compiling in Scratchbox and then installing it

```
$ sudo apt-get install cdbx (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)
```

Now you need to compile Qt BRisa typing(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 Qt BRisa on Maemo 5

To install Qt BRisa in Maemo 4 you just need to download the [libbrisa-dev\\_0.4.0maemo\\_all.deb](#) file and type the following command inside the folder you put it

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

Remember that you need to have libqxt installed (look the Compiling Qt BRisa in Maemo 5 (Via Scratchbox) session). The process is equal to the one in maemo4, but pay attention when making applications it's not the same thing.

## 3.4 windows — Compiling/Installing on Windows

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

### 3.4.1 Compiling Qt BRisa on Windows

The first thing you need to do is to install libqxt in your system. Qt BRisa repository has its own version of libqxt with some changes made to it in order to make things work fine in Brisa. You can download Qt BRisa folder [tree](#) and after that go to the libqxt directory (Qt BRisa/deps/libqxt/) using a command line terminal and type

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

This will install libqxt in your computer. Now you need to compile Qt BRisa project, by going to the main directory (Qt BRisa) inside Qt BRisa folder [tree](#) and typing

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

And Qt BRisa will be successfully compiled.

### 3.4.2 Installing Qt BRisa on Windows

After compiling Qt BRisa you just need to type the following command in order to install it

```
$ mingw32-make install
```

And Qt BRisa is finally installed in your computer.



---

# core-utils — Framework's core and utils

---

This section describes Qt BRisa BrisaCore and BrisaUtils namespaces. Each of the next sub sections will describe a class and how to use it.

## 4.1 config — Configuration facilities

Qt BRisa provides a very simple configuration API, just like BRisa Python does.

You can create Configuration Managers using the BrisaConfigurationManager class (from BrisaCore). Using this class you are able to set and get parameters in an easy way. Also, you can save the configuration and use it later.

Here is an example on how to use BrisaConfigurationManager class

### 4.1.1 Before writing your Qt BRisa applications

Whenever you are going to write a Qt BRisa application you need to change the .pro file in the project folder adding the following lines:

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

### 4.1.2 Using the Built-In Configuration Manager

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

// Generating my own configuration
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";
```

```
// Creating a BrisaConfigurationManager object using the generated configuration
BrisaConfigurationManager *myConfig = new BrisaConfigurationManager(configPath, state);

// Saving the config manager
myConfig->save();

// Checking brisaC++.owner parameter value and changing it
QDebug() << "value: " << myConfig->getParameter("brisaC++", "owner");
myConfig->setParameter("brisaC++", "owner", "newOwner");

myConfig->save();

// Updating the config
myConfig->update();

QDebug() << "value: " << myConfig->getParameter("brisaC++", "owner");
```

The output should be

```
value: "owner2"
value: "newOwner"
```

## 4.2 log — Logging facilities

Qt BRisa provides logging functions with a colored logging feature.

By using the log function, it's possible to create debug, warning, critical or fatal messages. The use is very simple and will be exemplified next

### 4.2.1 Global Logger

This module provides a global (or root) logger. It's simple to use, as you can see the the following example

```
#include <BrisaLog>

brisaLogInitialize()

QDebug() << "My debug message";
QWarning() << "My warning message";
QCritical() << "My critical message";
QFatal() << "My fatal message";
```

## 4.3 network — Network facilities

Qt BRisa has some functions which can simplify networking related tasks.

**getValidIP()**

You can use getValidIP function to retrieve an valid IP on any active network interface (eth0, wlan0 wlan1). The following example demonstrates how easy it is to use getValidIP function

```
#include <BrisaNetwork>
```

```
QString ip = getValidIP();
```

It will return a QString containing a valid IP

**getPort ()**

You can get a random port between the specified range of UpnP protocol using the getPort function. It will return a port number greater than 49152 and smaller than 65535 that is currently available

```
#include <BrisaNetwork>
```

```
quint16 port = getPort();
```

This will return a quint16 containing a port number.

## 4.4 webserver — Webserver facilities

### 4.4.1 Creating a simple Web Service using Qt BRisa

BRisa framework can be used to help you create your own web service. This section will demonstrate how to do this by showing the basics steps to create a simple web service. Although it is a simple web service, the following example covers the basics about using BRisa for this purpose.

We are going to use the following classes from the BrisaCore module:

- BrisaWebServer
- BrisaWebServiceProvider
- BrisaWebService

BrisaWebServer , BrisaWebServiceProvider and BrisaWebService classes are implemented using [libqxt](#), which has also other uses.

Let us begin by starting a Web Server creating an instance of BrisaWebServer, passing an IP address and a port number (you can use BrisaCore network functions to obtain these)

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

After that we need something to manage our services, that's where the BrisaWebServiceProvider comes in, we build it passing the BrisaWebServer we have just created.

```
BrisaWebServiceProvider *webserviceManager = new BrisaWebServiceProvider(webserver, this);
```

And then we call the “addService” method from BrisaWebService to add the *webserviceManager* to the root of the *webserver*, passing the desired url path. A BrisaWebServiceProvider itself is a webservice. It just doesn't emit the signals we want.

```
webserver->addService("hello", webserviceManager);
```

Now we can create the webservice (or webservices) passing our *webserver* to tie everything up.

```
BrisaWebService *hello = new BrisaWebService(webserver, this);
```

And after that we call the “addService” method from the *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”.

```
webserviceManager->addService("world", hello);
```

When a request arrives at our webservice two signals are emitted:

```
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 answer the request, simply call one of the “respond()” methods from `BrisaWebService`.

---

# upnp — UPnP modules

---

This section will describe UPnP module, mentioning its sub modules (packages), classes and its classes methods and also giving a brief description on how to use them.

## 5.1 device — Device building and deploying

Here we are going to describe Device package, present its classes and show how to use them.

This package contains two important classes to developers using Qt BRisa, these are the BrisaDevice and the BrisaService classes and will be discussed next.

### 5.1.1 device — BrisaDevice class

BrisaDevice class provides an easy and fast way to create UPnP devices. All you need to do is create a new BrisaDevice object and call its start() method to join the network and be visible to available control points.

#### Important Methods

BrisaDevice constructor which should receive all device information on its arguments.

```
void addEmbeddedDevice(const QString &deviceType = "",
                      const QString &friendlyName = "",
                      const QString &manufacturer = "",
                      const QString &manufacturerURL = "",
                      const QString &modelDescription = "",
                      const QString &modelName = "",
                      const QString &modelNumber = "",
                      const QString &modelURL = "",
                      const QString &serialNumber = "",
                      const QString &UDN = "",
                      const QString &UPC = "",
                      const QString &presentationURL = "");
```

Call this function passing the BrisaDevice to be embedded and the embedded device will be announced when the root device joins the network.

```
void addEmbeddedDevice(BrisaDevice *newEmbeddedDevice);
```

Method used to add a service to the device, you just need to pass a BrisaService object as the argument. The service will be automatically added to the device and the appropriate webserver urls paths will be created.

```
void addService(BrisaService *serv)
```

Call this method to join the network and start the device.

```
void start()
```

Stops the device and leaves the network sending ssdp messages for any embedded devices.

```
void stop()
```

### Creating a generic device

Here is a simple explanation on how creating a device and starting it. These are only basic steps to follow, you can check a more practical example at *Code Examples* section.

BrisaDevice constructor receives all information about a device, so it's a good idea to define the constants first.

```
#define DEVICE_TYPE "urn:schemas-upnp-org:device:MyDevice:1"
#define DEVICE_FRIENDLY_NAME "My Device Name"
#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 example device"
#define DEVICE_MODEL_NAME "My device model"
#define DEVICE_MODEL_NUMBER "1.0"
#define DEVICE_MODEL_URL "https://garage.maemo.org/projects/brisa"
#define DEVICE_SERIAL_NUMBER "1.0"
```

Then we can call the constructor passing the device informations we just defined

```
BrisaDevice myDevice(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,
                    getCompleteUuid());
// The getcompleteuuid() function returns a valid device udn automatically
```

We can now add our services to the device we just created (creation of services will be explained on the next section)

```
myDevice.addService(myService);
```

And finally we start our device

```
myDevice.start();
```

We're done! Now you know the basics steps of creating a device.

## 5.1.2 service — BrisaService class

BrisaService class allows you to create UPnP services in an easy way. You only need to create your actions and add them to the BrisaService object. After that your service can be linked to any device you want.

### Important Methods

BrisaService constructor. Receives service information.

```
BrisaService(const QString &serviceType,
             const QString &serviceId = "",
             const QString &scpdUrl = "",
             const QString &controlUrl = "",
             const QString &eventSubUrl = "",
             const QString &host = "",
             QObject *parent = 0);
```

Method to add an action to the service.

```
void addAction(BrisaAction *action);
```

This method returns the service's state variables list.

```
const QList<BrisaStateVariable *> getStateVariableList();
```

### Implementing a simple service.

This section will show an example on how to create a simple service.

First thing you need to do is to write your scpd.xml file, like this one:

```
<?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>actionName1</name>
      <argumentList>
        <argument>
          <name>ExampleInText</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>
    <action>
      <name>actionName2</name>
      <argumentList>
        <argument>
```

```

        <name>ExampleInText</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>
<action>
    <name>actionName3</name>
    <argumentList>
        <argument>
            <name>ExampleInText</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 make it child of BrisaService class. You will need to specify the service name, the service type and the scpd.xml file. Also you'll need to implement your service actions.

First, let's define the service informations we need

```

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

```

After that, let's include the BrisaService and the BrisaAction libraries to create our own service and actions

```

#include <BrisaAction>
#include <BrisaService>

using namespace BrisaUpnp;

```

To create your actions you just need to implement a method with a specific signature for each BrisaAction you want and they will be automatically bind to the service by the framework. You can create your actions as slots or as methods. Slots are automatically bound to the service but if you define your actions as methods you'll need to mark them as Q\_INVOKABLE as in the following example. Note that actions' names *must* be in lowercase.



```

class MyService : public BrisaService
{
    public:
        MyService() : BrisaService(SERVICE_TYPE,
                                    SERVICE_ID) { }

        Q_INVOKABLE
        BrisaOutArgument* actionname1(BrisaInArgument* const inArguments,
                                       BrisaAction* const action)
        {
            Q_UNUSED(action);
            BrisaOutArgument *outArgs = new BrisaOutArgument();
            QString inArg = inArguments->value("ExampleInText");
            outArgs->insert("TextOut", inArg + "Out!!");
            return outArgs;
        }

    private:
        Q_INVOKABLE
        BrisaOutArgument* actionname2(BrisaInArgument* const inArguments,
                                       BrisaAction* const action)
        {
            Q_UNUSED(action);
            BrisaOutArgument *outArgs = new BrisaOutArgument();
            QString inArg = inArguments->value("ExampleInText");
            outArgs->insert("TextOut", inArg + "Out!!");
            return outArgs;
        }

    private slots:
        BrisaOutArgument* actionname3(BrisaInArgument* const inArguments,
                                       BrisaAction* const action)
        {
            Q_UNUSED(action);
            BrisaOutArgument *outArgs = new BrisaOutArgument();
            QString inArg = inArguments->value("ExampleInText");
            outArgs->insert("TextOut", inArg + "Out!!");
            return outArgs;
        }
};

```

Now that we have our service class we can create a service from it in the following way

```

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

```

When we add this service to some device and call device's start method all attributes of the service are going to be initialized.

## 5.2 control\_point — Control Point API

Here you can learn how to use the Control Point package API to subscribe to unicast eventing

### 5.2.1 service — BrisaControlPointService class

#### Subscribing to 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 passing the subscription timeout as in the following code

```
class ControlPoint : BrisaControlPoint

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

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

    void unsubscribe(BrisaControlPointService *service);

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

Now implement these methods and the slot on the source file

```
void ControlPoint::subscribe(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]];
    }
}
```

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 — Simple Service Discovery Protocol API

Here you are going to find the description of the BrisaSSDPService and BrisaSSDPClient classes used by devices and control points for advertising and discovery of network services, respectively.

### 5.3.1 ssdpserver — SSDP Server implementation

Qt BRisa provides a `BrisaSSDP`Server class which can be used by a device for announcing its presence in the network, its embedded devices and all services provided by it

#### BrisaSSDP

Server class

Implementation of a SSDP Server.

#### Attributes

The attribute of the `BrisaSSDP`Server class are:

- `running` - Boolean variable that holds server running state
- `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 `BrisaSSDP`Server 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 `BrisaSSDP`Server.

##### `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 goodbye message to the multicast group with the given information.

##### `datagramReceived()`

This slot is called when the `readyRead()` signal is emitted 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

Qt BRisa provides a `BrisaSSDP`Client class which can be used by control points for receiving notification messages

## BrisaSSDPClient class

Implementation of a SSDP Client.

### Attributes

The attribute of the BrisaSSDPClient class are:

- `running` - Boolean variable that holds server running state
- `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()”.

---

# Code Examples

---

Here we're going to show some examples that will help you understand better how to make applications using Qt BRisa.

## 6.1 Creating a Binary Light Device

This is the implementation of a Binary Light Device, which specifications you can find [here](#).

Before creating a Device it's good to create the service(s) it's going to use. So we're going to create a service called SwitchPower before creating the BinaryLightDevice.

Let's define the service's information at 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 that, let's include the Brisa libraries that are used to create the Actions and the services.

```
#include <BrisaAction>
#include <BrisaService>

using namespace BrisaUpnp;
```

Now we should start implementing our SwitchPower service

```
SwitchPower() : BrisaService(SERVICE_TYPE,
                             SERVICE_ID,
                             SERVICE_XML_PATH,
                             SERVICE_CONTROL,
                             SERVICE_EVENT_SUB) { }

BrisaOutArgument* settarget(BrisaInArgument* const inArguments, BrisaAction* const action) {
    getStateVariable("Target")->setAttribute(BrisaStateVariable::Value,
                                              inArguments["NewTargetValue"]);
    getStateVariable("Status")->setAttribute(BrisaStateVariable::Value,
                                              inArguments["NewTargetValue"]);
}
```

```

    BrisaOutArgument *outArgs = new BrisaOutArgument();
    return outArgs;
}

BrisaOutArgument* gettarget(BrisaInArgument* const inArguments, BrisaAction* const action) {
    Q_UNUSED(inArguments)

    BrisaOutArgument *outArgs = new BrisaOutArgument();
    outArgs->insert("RetTargetValue", getStateVariable("Target")
        ->getAttribute(BrisaStateVariable::Value));

    return outArgs;
}

BrisaOutArgument* getstatus(BrisaInArgument* const inArguments, BrisaAction* const action) {
    Q_UNUSED(inArguments)

    BrisaOutArgument *outArgs = new BrisaOutArgument();
    outArgs->insert("ResultStatus", getStateVariable("Status")
        ->getAttribute(BrisaStateVariable::Value));

    return outArgs;
}

```

Now it's time to start implementing our device. Let us include BrisaDevice class and our service header, define the namespace and our device informations.

```

#include <BrisaDevice>
#include "switchPower.h"

using namespace BrisaUpnp;

#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"

```

It's time we start implementing our device! We need to create our header file with the slots, attributes and methods we are going to use. In this example our device will be a QWidget that contains a BrisaDevice and two BrisaStateVariable objects as attributes, but if you don't want to display a graphical interface to represent your device you can just make it child of BrisaDevice (instead of having a BrisaDevice 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;

};

```

Now, our header file is ready and we have to start the implementation of the device class source file. Here's its constructor

```

Device::Device(QWidget *parent) : QWidget(parent)
{
    this->binaryLight = new 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,
                                        getCompleteUuid());

    SwitchPower *switchPower = new SwitchPower();
    switchPower->setDescriptionFile("SwitchPower-scpd.xml");
    this->binaryLight.addService(switchPower);
    this->binaryLight.start();

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

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

```

And now we just need to implement our statechanged slot.

```

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

```

Our device is done and the last thing we need to do is initialize the device in the main file.

```

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

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Device dev;
    dev.show();

    return a.exec();
}

```

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 download 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 command line Control Point for Binary Light devices, but this will cover the basics of creating a control point. This Control Point has the following features:

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

As we said 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 include the libs we are going to use in our code.

```
#include <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>
```

Because our control point is interacting with Binary Light devices, we need to define the device type we are communicating with and service that contains the actions we are going to use

```
//The binary light type
#define BINARY_LIGHT_TYPE      "urn:schemas-upnp-org:device:BinaryLight:1"

//Service that contains the actions we'll use
#define SERVICE_SWITCH_POWER  "urn:schemas-upnp-org:service:SwitchPower:1"
```

Now let's create and implement the ControlPoint class, we decided to make it a BrisaControlPoint itself, but you could have a BrisaControlPoint object to work as a control point in another class of your choice if you'd want to.

Our Control Point class will show when devices enter or leave the network, list devices to the user and it is going to interact with the devices by using the actions provided by them. In order to implement these features we are going to need the following attributes, methods and slots

```
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;
    // Gets the SetTarget response
    void setTargetResponse(QString response, QString method);
    // Gets the GetTarget response
    void getTargetResponse(QString response, QString method);
    // Gets the GetStatus response
    void getStatusResponse(QString response, QString method);
    //Slot used when a device joins the network.
    void onNewDevice(BrisaControlPointDevice *dev);
    //Slot used when a device leaves the network.
    void onRemovedDevice(QString desc);
    //Slot used to handle error
    void requestError(QString errorMessage, QString methodName);
};

```

Our Control Point header file is ready, so let us implement all methods and slots that were declared. The first important thing to do is implementing the constructor method that is going to create all the necessary event connections and start both the Device's discovery and the Thread commands.

```

#include "controlpoint.h"

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

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

    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 simple 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(BrisaControlPointDevice::Udn))
            return;
    }
    devices.append(device);
    device->downloadIcons();
    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;
    }
}
```

Before continuing let us see quickly how to download device's icons if you need them in your control point.

BrisaControlPointDevice class has a method called downloadIcons which request the download of all its icons. When all downloads are finished BrisaControlPointDevice will emit a onReadyDownloadIcons(BrisaControlPointDevice\*) signal. So basically all you need to do is the following:

Create a slot which you are going to use to do whatever you want with devices icons. I'm calling my slot iconsDownloadFinished, but you can put any name you want. Inside your Control Point header file:

```
public slots:
    void iconsDownloadFinished(BrisaControlPointDevice*)
```

And inside your source file call BrisaControlPointDevice downloadIcons() method when you find a device. Also, connect onReadyDownloadIcons(BrisaControlPointDevice\*) signal to the slot you created, like this:

```
device->downloadIcons();
connect(device, SIGNAL(onReadyDownloadIcons(BrisaControlPointDevice*)),
        this, SLOT(iconsDownloadFinished(BrisaControlPointDevice*)))
```

All BrisaIcons objects inside your BrisaControlPointDevice will now have a QIcon object with the icon's object. To access BrisaIcons objects you can use device getIconList() method and then BrisaIcon getIcon() to retrieve QIcon object related to that BrisaIcon!

```
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();
    QApplication::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];

```

Each of the action call functions (turn on, turn off, getTarget, getStatus) need a slot to receive the action call response. The call itself is done in a simple way directly from the BrisaControlPointService.

```

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(BrisaOutArgument, QString)), this,
            SLOT(getTargetResponse(BrisaOutArgument, QString)));
    connect(service, SIGNAL(requestError(QString, QString)), this,
            SLOT(requestError(QString, 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("1")){
            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(BrisaOutArgument, QString)), this,
            SLOT(getStatusResponse(BrisaOutArgument, QString)));
    connect(service, SIGNAL(requestError(QString, QString)), this,
            SLOT(requestError(QString, QString)));
    service->call("GetStatus", param);
}

void ControlPoint::getStatusResponse(BrisaOutArgument 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(BrisaOutArgument, QString)), this,
            SLOT(setTargetResponse(BrisaOutArgument, QString)));
    connect(service, SIGNAL(requestError(QString, QString)), this,
            SLOT(requestError(QString, 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(BrisaOutArgument, QString)), this,
            SLOT(setTargetResponse(BrisaOutArgument, QString)));
    connect(service, SIGNAL(requestError(QString, QString)), this,
            SLOT(requestError(QString, 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;
    }
}

void ControlPoint::requestError(QString errorMessage, QString methodName)
{
    qDebug() << errorMessage + " when calling " + methodName;
}

```

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 passed to the ControlPoint to perform the actions

```

class HandleCmds : public QThread
{
    Q_OBJECT
public:
    void run() {
        QTextStream stream(stdin);
        do{
            QString line;
            QApplication::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();
};

```

We already connected it in the Control Point when we were creating the control point constructor

```
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();
```

So our control point is done. We only need our main now

```
#include <QCoreApplication>
#include "controlpoint.h"

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    a.processEvents();
    ControlPoint cp;

    return a.exec();
}
```

And that's how we create a simple control point using Qt BRisa.

**You can find the code in:** [https://garage.maemo.org/plugins/scmsvn/viewcvs.php/qt/trunk/qt-brisa/doc/docSource/example\\_code/ControlPoint/?root=brisa](https://garage.maemo.org/plugins/scmsvn/viewcvs.php/qt/trunk/qt-brisa/doc/docSource/example_code/ControlPoint/?root=brisa). Just download ControlPoint.zip file.

---

# Wizard — Creating a BRisa Project

---

## 7.1 About Qt BRisa Wizard Application

Qt BRisa Wizard Application is capable of creating Devices and ControlPoints projects using only a few information about device and service (e.g. device's constants, services' actions, actions arguments).

User should inform the project and device names.

The name you choose for the project will be the name of the directory where Devices and ControlPoints directories related to this project will be created.

The user doesn't need to inform the controlPoint's name, it will be defined as the following:

```
device_name + "ControlPoint"
```

When in description page, user will need to inform the required constants values or accept default ones.

In Services Definition page, the user may inform which services will be added to the device and Qt BRisa wizard application will create a header, a source and a xml file for each service.

It is not allowed that any action has arguments or stateVariables names repeated when defining its parameters. However, distinct actions may have the same argument or stateVariable name (as in the Calculator example).

The field DefaultValue is not mandatory. This line will be ignored in xml file if not defined.

The modification of Type or DefaultValue from some StateVariable used in distinct actions causes that modification to happen in all actions that make use of this StateVariable. This happens because the StateVariable belongs to Service, not to a specific action.

The generated code is currently compiled by Qt 4.6.2 version and QtCreator version 2.0.1 and earlier.

It's possible to automatically repeat the arguments and stateVariables from one action to all other actions using the button "repeat arguments". Any arguments or stateVariables defined in the others actions, will be lost and replaced by the arguments and stateVariables in current action.

Once compiled, the ControlPoint when executed, sends a test string with a value 1 for each input argument of the first action and sends to the device, which responds to the concatenation of these strings to ControlPoint.

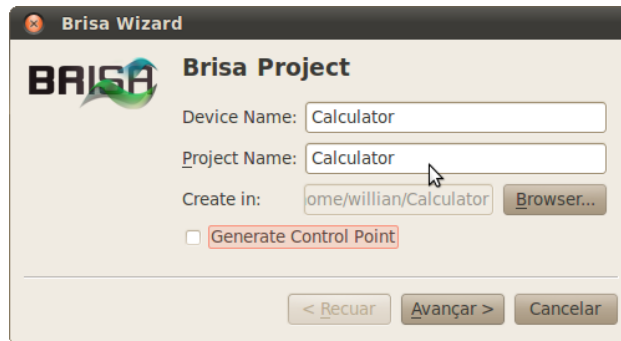
If the first action has no input arguments, the device sends a string test for the ControlPoint with the following format: "testing action" + action\_name.

If the first action has no output arguments, then it does not respond to ControlPoint, it'll only warn to debug that the input arguments have changed.



## 7.2 How create a Brisa Project

On the first page, as you type the device name, the project name will be completed with the same name of device, as shown below:



The dialog box titled "Brisa Wizard" shows the "Brisa Project" configuration. It includes the following fields and options:

- Device Name:** Calculator
- Project Name:** Calculator
- Create in:** /home/willian/Calculator
- ☐ **Generate Control Point**

At the bottom, there are three buttons: "< Recuar", "Avançar >", and "Cancelar".

If desired, you can change the name of the Project, as shown below:



The dialog box titled "Brisa Wizard" shows the "Brisa Project" configuration with the following changes:

- Device Name:** Calculator
- Project Name:** CalculatorProject
- Create in:** /home/willian/CalculatorProject
- ☒ **Generate Control Point**

The buttons at the bottom remain the same: "< Recuar", "Avançar >", and "Cancelar".

If you do not need to generate a Control Point for the Device, leave unchecked the option to Generate Control Point. The project by default is saved in the user's home directory, but if wanted the path can be changed by clicking on the "Browse..." button.

When the "next" button is clicked, the second page will appear asking for device information:



The dialog box titled "Brisa Wizard" shows the "Device" configuration page. It includes the following fields:

- Device Type:** urn:schemas-upnp-org:device:Calculator:1
- Friendly Name:** Calculator Device
- Manufacturer Name:** Brisa Corp.
- Manufacturer Url:** https://garage.maemo.org/projects/brisa
- Model Description:** An UPnP Calculator Device
- Model Name:** Calculator Device
- Model Number:** 1.0
- Model Url:** https://garage.maemo.org/projects/brisa
- Device Serial Number:** 1.0

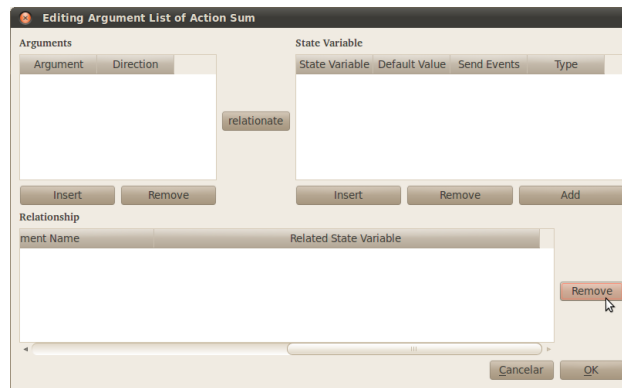
At the bottom, there are three buttons: "< Recuar", "Avançar >", and "Cancelar".

Based on what the user typed into the first page, suggestions are automatically generated for the fields in this page, if desired, the user can change them. This information will form the Calculator.h file's constants, which will be generated when the wizard is finished.

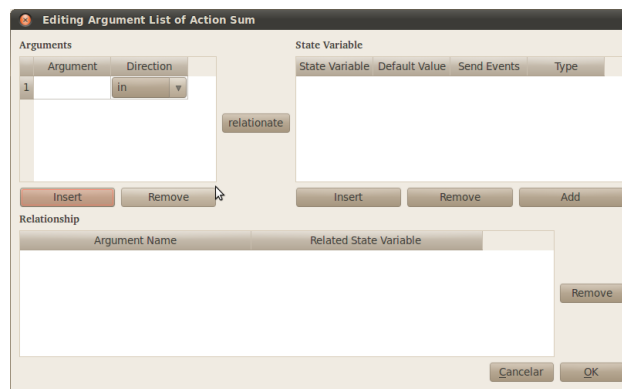
After inputting the device information you can click “next” button and go to third page where you’ll be requested for information about the services that are part of the Device. And again there will be suggested values for each field of each new service added. At the end each service will be related to one source, one header and one xml file.

In this demonstration, only the service “CalculatorActions.” was added. Advancing to the next page we inform the actions that constitute the Device Calculator:

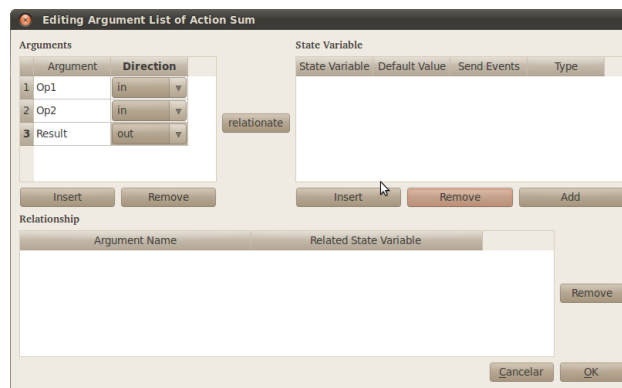
With all actions added, the next step is to define the arguments that each action. Double click on an action or select a desired action and click on the “Edit Arguments” button, the follow screen will appear:



To insert a new argument click on the “Insert” button and a new line will appear on the list:



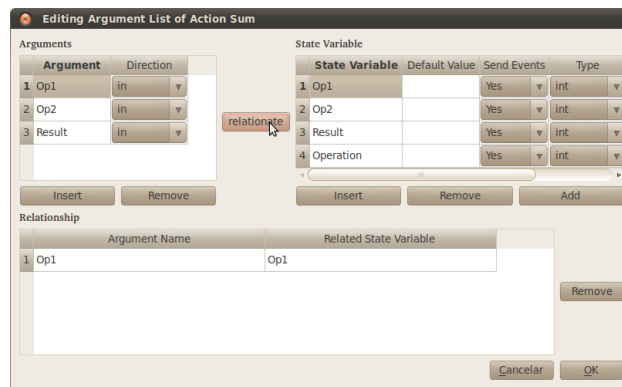
In this example we define two input arguments “Op1” and “Op2” and an output: “Result”.



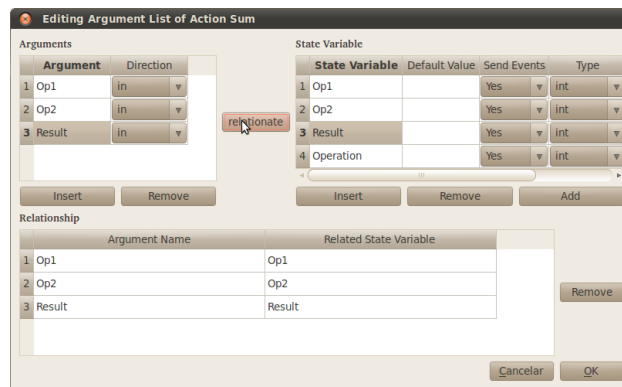
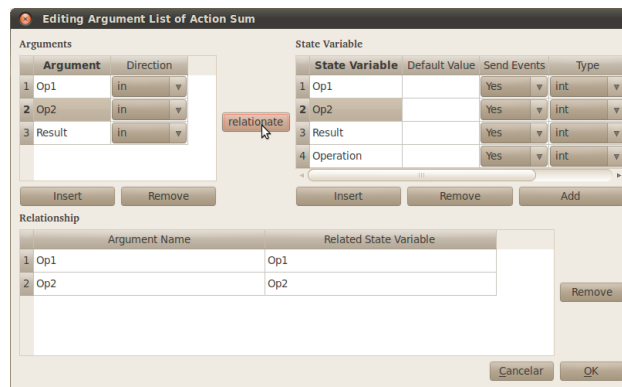
In next step we define the state variable with same name of the arguments. The field “Default Value” is optional, fill it in case you wish to have a default value of the corresponding state variable.



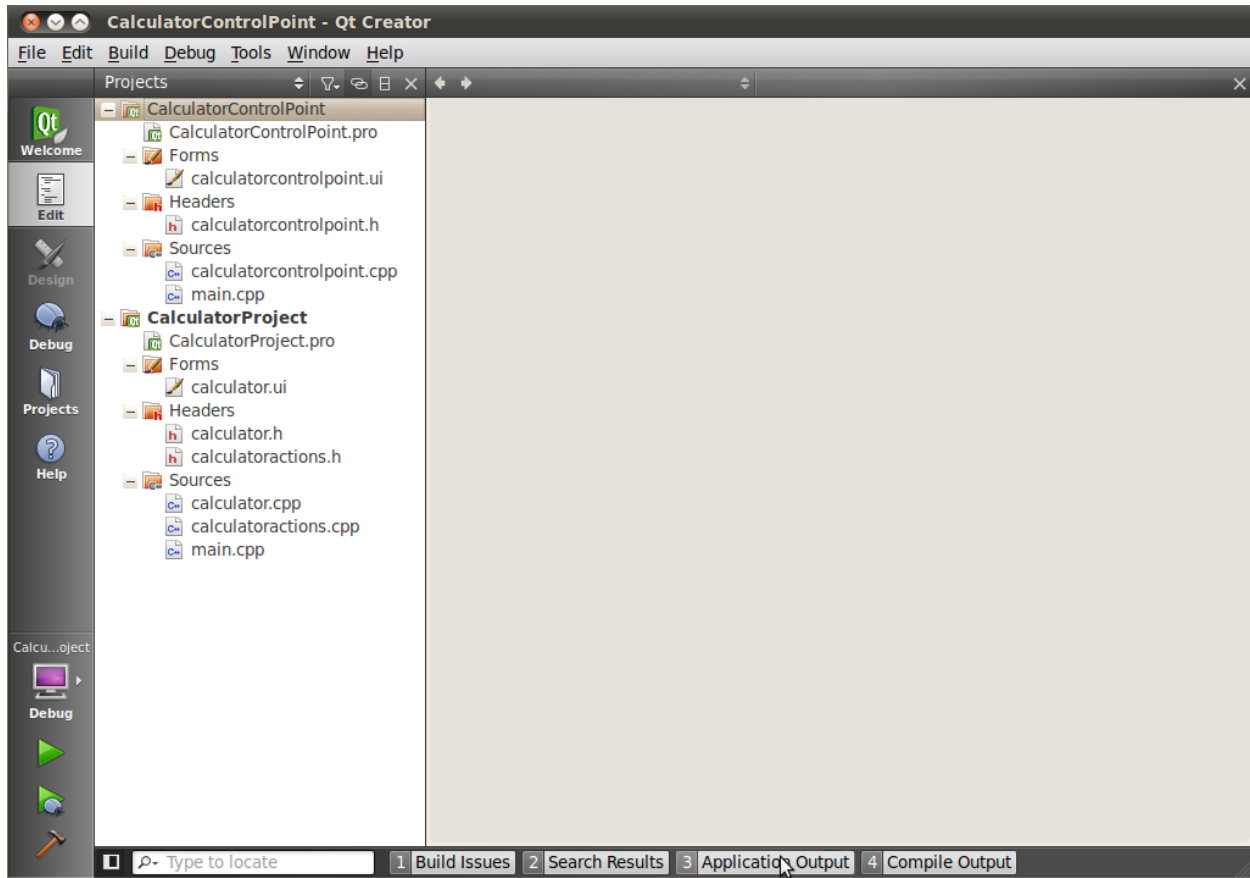
In next step we can link the arguments to state variable:



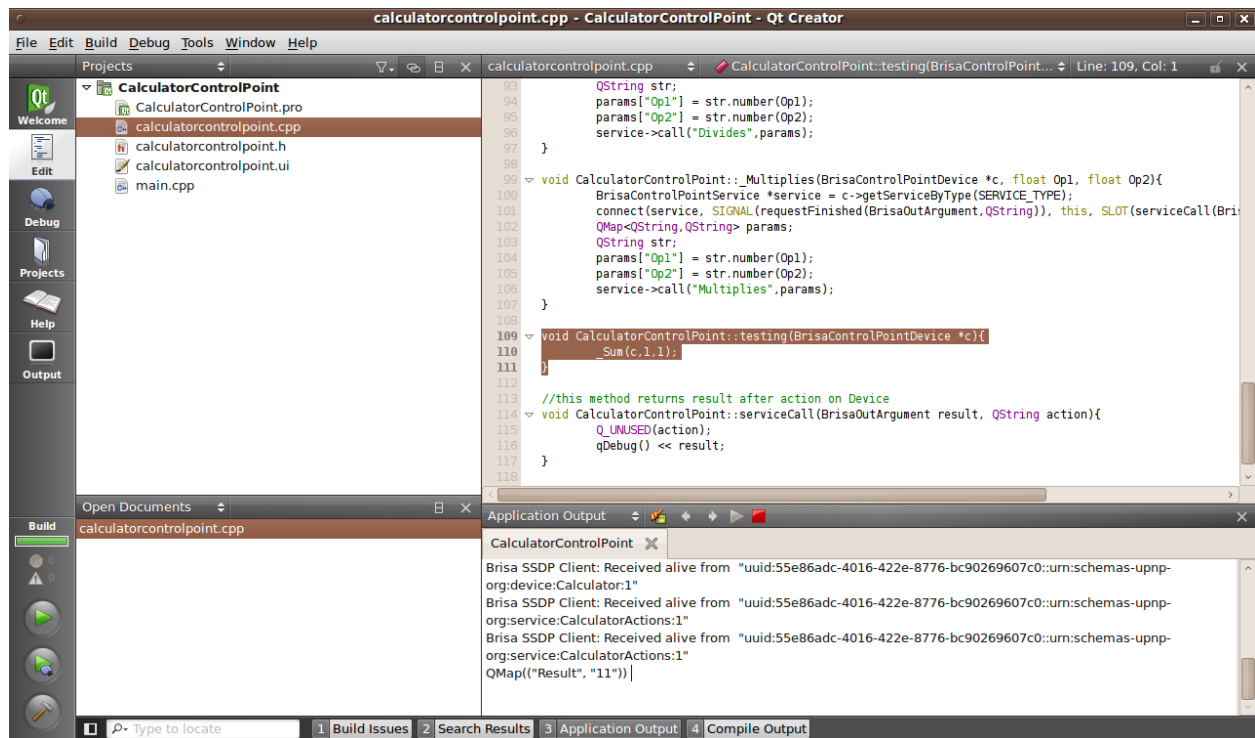
And repeat the same procedure to others arguments and state variables, as shown next.



And we are done creating our calculator device. As we chose to generate a ControlPoint in the beginning of the wizard, two QtCreator projects will be opened at the end, as shown:



These projects are buildable. Running ControlPoint, will perform the action Sum, which is the first action defined by setting the input arguments to "1". On the application output appears that the Device variables "Op1", "Op2" were changed to the value 1 and the variable "Operation" was changed to "Sum". Then, the ControlPoint receives the concatenation of the values of "Op1" and "Op2" as the answer "11". Check the implementation of projects and ControlPoint Device:



---

# Copyright

---

BRisa and this documentation is:

Copyright (C) 2007-2010 BRisa Team <[brisa-develop@garage.maemo.org](mailto:brisa-develop@garage.maemo.org)>

---

# Indices and tables

---

- *genindex*
- *modindex*
- *search*



---

# Index

---

## A

[addAction](#), 12  
[addEmbeddedDevice](#), 10  
[addService](#), 11

## B

[BRisa Project](#), 1  
[BrisaCore](#), 6  
[BrisaService](#), 12

## C

[control point](#), 1

## D

[datagramReceived\(\)](#) (built-in function), 16, 17  
[device](#), 1  
[doByeBye\(\)](#) (built-in function), 16

## G

[getPort\(\)](#) (built-in function), 8  
[getStateVariableList](#), 12  
[getValidIP\(\)](#) (built-in function), 7

## I

[isRunning\(\)](#) (built-in function), 16, 17

## L

[libqxt](#), 3  
[logger](#), 7

## N

[network](#), 7  
[notifyReceived\(\)](#) (built-in function), 17

## Q

[Qt BRisa](#), 1

## S

[service](#), 1  
[start](#) (BrisaDevice method), 11  
[start\(\)](#) (built-in function), 16, 17  
[stop](#) (BrisaDevice method), 11  
[stop\(\)](#) (built-in function), 16, 17

## U

[UPnP](#), 1

## W

[webserver](#), 8