English

Asema IoT Central

Raspberry Pi example manual

ASEMA

Table of Contents

1. Introduction

The Raspberry Pi sample is free sample that allows you to test the functionality of Asema IoT Central and Asema IoT Edge with real hardware by connecting a Raspberry Pi computer to it.

The sample uses Asema IoT Edge software to turn a Raspberry Pi into an IoT edge device. Effectively you can use the device as a gateway to some sensoring or control device connected to the Raspberry Pi board. Asema IoT Edge software itself is free to download and install on the Pi but it has a license manager that requires it to find a valid Asema IoT Central instance where it can register itself. Once registration is done, the license manager unlocks the features of Asema IoT Edge. The free demo version of Asema IoT Central allows connecting one Asema IoT Edge instance to it and validating the license. So if you do not yet have an installation of Asema IoT Central, you can use the demo version for this sample purpose.

To run the sample you need:

- One Asema IoT Central instance running somewhere in the network
- The Asema IoT Edge binary installer
- Qt libraries for Raspberry Pi

Please see Asema IoT Central manual for instructions on how to download and install Asema IoT Central. Once done, the next chapters in this manual show how to connect an Asema IoT Edge to it.

2. Configuring and installing

2.1. Download and install the Asema IoT Edge binary

You can download the binary package for Asema IoT Edge from https://iot.asema.com/iotc/downloads.html

Once you have the installer (.deb) package, transfer it to your Edge (if not already downloaded with it) and then install it using the package manager:

```
sudo dpkg -i asema-edge_1.0-1.deb
```

2.2. Build and install Qt

To run Asema IoT Edge on Raspberry Pi, you need to have the Qt library packages installed on the Raspberry Pi. The libraries needed to run the software are already included in the binary installer, so if you just want to run Asema IoT Edge, the following steps are not necessary. However, if you create custom drivers, you need to compile them and for that purpose you need the toolchain. In this case it may be necessary to have the full Qt development kit compiled on your Raspberry Pi.

Important
Compiling Qt is only necessary if you do custom plugin development on your Raspberry Pi. Just running Asema IoT Edge does not require these steps.

Because there are no official packages for recent versions of Qt for Raspberry Pi, it needs to be compiled manually. The following describes the steps to build Qt 5.11.1 for Raspberry Pi with all bells and whistles required in development work.

2.2.1. Acquire Raspbian development image

Create a build directory and download the latest Raspbian development image from raspberrypi.org:

```
mkdir ~/raspi_build
cd ~/raspi_build
wget http://downloads.raspberrypi.org/raspbian_latest -O wheezy-raspbian-latest.zip
unzip wheezy-raspbian-latest.zip
```

2.2.2. Add SSL and Bluetooth support

To do this, you need a SD card of at least 8 GB because the development image won't fit on a 4 GB card when openssl packages are installed. Using a card larger than 8 GB is also not recommended, because the resulting image file will have the same size as the card used.

First, copy the Raspbian image to the FAT32-formatted 8GB SD card. Replace the SD_DEVICE value with the device that corresponds to the SD card (without any trailing digits). Replace the IMAGE_FILE value with the name of the latest image file.

```
SD_DEVICE=/dev/sdx
IMAGE_FILE=2018-06-27-raspbian-stretch.img
sudo dd bs=1M if=${IMAGE_FILE} of=${SD_DEVICE} conv=fsync
```

Then, insert the SD card to the Raspberry Pi, boot the device and connect it to the Internet. Note that the development image uses by default DHCP to get an IP address. If you do not have DHCP available, configure the network manually. To do this, edit /etc/dhcpcd.conf. Uncomment the "Example static IP configuration" section and set `routers` and `domain_name_servers` to match the settings in your network. Then set the IP address, any unused IP should be fine.

Next, edit /etc/apt/sources.list and uncomment the `deb-src` line. Then, run updates and install the development packages needed for adding openssl, Bluetooth and xcb support to the Qt build:

```
sudo apt update
sudo apt build-dep qt4-x11
sudo apt build-dep libqt5gui5
sudo apt install libssl-dev libbluetooth-dev libudev-dev libinput-dev libts-dev libxcb-xinerama0-dev libxcb-xinerama0
```

Shut down the Raspberry Pi, remove the SD card and insert it in your computer to create a new image.

```
sudo dd bs=1M if=${SD_DEVICE} of=raspbian-openssl.img
```

2.2.3. Mount the Raspbian development image

First, find out the offset value needed for mounting the image. Run the following command:

```
sudo fdisk -l raspbian-openssl.img
```

This should output something like the following:

```
Disk raspbian-openssl.img: 7.3 GiB, 7826571264 bytes, 15286272 sectors
  Units: sectors of 1 * 512 = 512 bytes
  Sector size (logical/physical): 512 bytes / 512 bytes
  I/O size (minimum/optimal): 512 bytes / 512 bytes
  Disklabel type: dos
  Disk identifier: 0x2e64e5ed

  Device                Boot Start      End  Sectors  Size Id Type
  raspbian-openssl.img1       8192    96663    88472 43.2M  c W95 FAT32 (LBA)
  raspbian-openssl.img2      98304 15286271 15187968  7.2G 83 Linux
```

Multiply the start of the second sector (98304 in this case) by the sector size (512 in this case). Use the result (50331648) in the mount command as follows:

```
sudo mkdir /mnt/rasp-pi-rootfs
sudo mount -o loop,offset=50331648 raspbian-openssl.img /mnt/rasp-pi-rootfs
```

2.2.4. Get the cross-compiler toolchain

Go to https://osdn.net/projects/sfnet_rfidmonitor/, download the file named gcc-4.7-linaro-rpi-gnueabi-hf.tbz, copy it to ~/raspi_build directory and unpack it:

```
tar xvf gcc-4.7-linaro-rpi-gnueabihf.tbz
```

Get cross-compile tools and check out the correct revision that supports specifying the toolchain in the command line (in newer ones this feature has been removed).

```
git clone https://github.com/darius-kim/cross-compile-tools.git
cd cross-compile-tools
git checkout d49f517eaa3bc819f78a01a5eead9e1a697ce6f5
cd ..
```

Get raspberrypitools:

```
git clone https://github.com/raspberrypi/tools.git
```

Adjust symbolic links in rootfs libraries to be relative:

```
mkdir sysroot sysroot/usr sysroot/opt
cp -a /mnt/rasp-pi-rootfs/lib/ sysroot
cp -a /mnt/rasp-pi-rootfs/usr/include sysroot/usr/
cp -a /mnt/rasp-pi-rootfs/usr/lib sysroot/usr/
cp -a /mnt/rasp-pi-rootfs/opt/vc sysroot/opt/
wget https://raw.githubusercontent.com/riscv/riscv-poky/priv-1.10/scripts/sysroot-relativelinks.py
chmod +x sysroot-relativelinks.py
./sysroot-relativelinks.py sysroot
```

2.2.5. Compile Qt5

Get Qt5 sources. The following checks out version 5.11.1. If you want something else, replace the version tag in the `git checkout` command.

```
git clone git://code.qt.io/qt/qt5.git
cd qt5
git checkout v5.11.1
perl init-repository
```

Patch the source next. Because this compile method uses gcc 4.7, a patch is needed to make the Qt build work:

```
cd qtlocation
wget https://bugreports.qt.io/secure/attachment/74716/patch-src_location_declarativemaps_qdeclarativegeomap.cpp
patch -p0 < patch-src_location_declarativemaps_qdeclarativegeomap.cpp
cd ..
```

Finally, compile Qt5. Run configure and make as follows:

```
  ./configure -opengl es2 -device linux-rasp-pi-g++ -device-option CROSS_COMPILE=~/raspi_build/tools/arm-bcm2708/gcc-linaro-
arm-linux-gnueabihf-raspbian-x64/bin/arm-linux-gnueabihf- -sysroot ~/raspi_build/sysroot -opensource -confirm-license -reduce-
exports -no-use-gold-linker -no-gbm -release -make libs -prefix /usr/local/qt5pi -extprefix ~/raspi_build/qt5 -hostprefix ~/
raspi_build/qt5 -v
  make -j4
  make install
```

Now the Qt5 libraries should be installed in the Raspbian development image and cross-compiling Qt apps for Raspberry Pi can be done by using `$HOME/raspi_build/qt5/bin/qmake` to build projects.

2.3. Start Asema IoT Edge

Once the Qt libraries are installed, you can start Asema IoT Edge. To do so, simply open a terminal on the Raspberry Pi and issue:

```
> asema_iot_edge
```

Or for headless mode:

```
> asema_iot_edge_headless
```

You can now access the web admin interface of Asema IoT Edge. It is by default found at http://[insert_ip_of_raspberry_pi_here]:8080. To login, the default username and password are "admin" and "admin".

3. Creating hardware drivers

Now that you have an Asema IoT Edge running, the next step is to do something real with it. This functionality is implemented with drivers.

3.1. Programming drivers for Raspberry Pi

A hardware driver for Raspberry Pi is an Asema IoT Edge plugin. The IoT system loads this plugin at startup and you can assign it to the objects and properties to link it to IoT applications.

Plugins are programmed in C++ using the Qt library. Instructions in the previous chapter show how to compile the Qt packages for Raspberry Pi. As a result, you will also have the toolchain needed to compile the drivers (namely qmake, make and the associated compiler and linker).

To learn in general how plugins are programmed and the anatomy of a hardware driver for Asema IoT Edge and Asema IoT Central, read the Asema IoT Plugin Interface 1.0 manual.

3.2. A sample driver

This sample shows basic hardware hacking by connecting something to the GPIO ports of the Raspberry Pi and programming a driver to control those ports.

Because of the object-property concept and the free driver module programmability, you can basically implement any hardware function the Raspberry Pi hardware supports. To keep this example simple and short enough for the manual, we'll just do some LED blinking.

To simplify the programming of GPIO pins on Raspberry Pi, this sample uses the Wiring Pi library. To make this sample work, download and install that library from http://wiringpi.com.

Next, you need a project file for the driver. Below is a sample:

```
QT += core qml

TARGET = raspberryleddriver
TEMPLATE = lib

DEVEL_LIBS_HEADERS = "./developerlibs/1.0/"

QMAKE_CXXFLAGS += -std=gnu++11

INCLUDEPATH += $${DEVEL_LIBS_HEADERS}

SOURCES += $${DEVEL_LIBS_HEADERS}/hardware/hardwarecontroller.cpp \
 RaspberryGpioLedDriver.cpp

HEADERS += $${DEVEL_LIBS_HEADERS}/plugin/plugin.h \
 $${DEVEL_LIBS_HEADERS}/plugin/customhwdriverplugin.h \
 $${DEVEL_LIBS_HEADERS}/hardware/hardwarecontroller.h \
 $${DEVEL_LIBS_HEADERS}/interfaces/interfacecapability.h \
 RaspberryGpioLedDriver.h

LIBS += -lwiringPi

DESTDIR = lib
OBJECTS_DIR = objects
MOC_DIR = MOCs
```

Once you have the C++ files of the driver, run `qmake` with this file to get the makefile to compile your driver. Then compile and install it on your Rasperry Pi.

Next, the driver itself. The header for it is below. Most of the methods supported by a hardware driver remain unimplemented, in this sample we just implement a couple of them. The two primary methods to implement are `getCapabilities()` and `setDeviceProperty()`. The former gives us the correct

settings for the Asema IoT Edge user interfaces, the latter provides the actual GPIO pin control functionality.

```
#ifndef __RASPBERRYLEDDRIVER_H
#define __RASPBERRYLEDDRIVER_H

#include "plugin/customhwdriverplugin.h"

class RaspberryGpioLedDriver  : public CustomHwDriverPlugin
{
 Q_OBJECT

public:
 RaspberryGpioLedDriver();

 const char* getName() { return "RaspberryLedController"; }
 QString getAddress() { return QString(); }

 bool initDriver() { return true; }
 void setSettings(QVariantMap settings) { Q_UNUSED(settings); }
 bool write(QVariant address, QByteArray data);
 QByteArray read(QVariant address) { Q_UNUSED(address); return QByteArray(); }

 int calibrateDevice(int calibrationType) { Q_UNUSED(calibrationType); return 0; }
 int locateDevice() { return 0; }
 int pingDevice() { return 0; }
 int restartDevice() { return 0; }
 int selfDiagnoseDevice() { return 0; }

 QVariant getDeviceProperty(QVariant propertyId) { Q_UNUSED(propertyId);  return QVariant(); }
 int requestDeviceProperty(QVariant propertyId) { Q_UNUSED(propertyId);  return 0; }
 int setDeviceProperty(QVariant propertyId, QVariant propertyValue);

 int requestDeviceConnectionState() { return 0; }
 QVariant getDeviceParameter(QVariant parameterId) { Q_UNUSED(parameterId); return QVariant(); }
 QVariant getDeviceState(QVariant stateId) { Q_UNUSED(stateId); return QVariant(); }
 void getDeviceStatus() {}
 const QMap<QString, QString> getParametersOfDevice() { return QMap<QString, QString>(); }

 int requestDeviceConnectionQuality() { return 1; }
 int requestDeviceFirmwareVersion() { return 0; }
 int requestDeviceClock() { return 0; }
 int requestDeviceData(QVariant dataId) { Q_UNUSED(dataId);  return 0; }
 int requestDeviceState(QVariant stateId) { Q_UNUSED(stateId); return 0; }
 int requestDeviceStatus() { return 0; }
 int requestDeviceValue() { return 0; }
 int requestParametersFromDevice() { return 0; }

 int setDeviceCalibrationParameters(QByteArray calibrationData) { Q_UNUSED(calibrationData); return 0; }
 int setDeviceClock(quint64 mseconds_from_epoch) { Q_UNUSED(mseconds_from_epoch); return 0; }
 int setDeviceState(QVariant stateId, QVariant stateValue) { Q_UNUSED(stateId); Q_UNUSED(stateValue); return 0; }
 int setDevicePermanentState(QVariant stateId, QVariant stateValue) { Q_UNUSED(stateId); Q_UNUSED(stateValue); return 0; }
 int setDeviceValue(QVariant value) { Q_UNUSED(value); return 0; }
 int setParametersToDevice(QMap<QString, QString> attributeData) { Q_UNUSED(attributeData); return 0; }

 int requestDeviceParameter(QVariant parameterId) {  Q_UNUSED(parameterId); return 0; }
 int setDeviceParameter(QVariant parameterId, QVariant parameterValue) {  Q_UNUSED(parameterId); Q_UNUSED(parameterValue);
 return 0; }
 int setDevicePermanentParameter(QVariant parameterId, QVariant parameterValue) {  Q_UNUSED(parameterId);
 Q_UNUSED(parameterValue); return 0; }

 int unpairDevice() { return 0; };

 const QList<InterfaceCapability> getCapabilities();

public slots:
 void discoverAndConnectToDevice() { connectToDevice(); }
 void connectToDevice() { connected = true; emit deviceConnected(); }
 void disconnectFromDevice() { connected = false; emit deviceDisconnected(); }


private:
 void switchLed(bool direction);
 void setGpioPort(int portNumber, int portValue);

};

#endif // __RASPBERRYLEDDRIVER_H
```

So, that was the header. Now to the actual beef, the methods. Below is the cpp file for our driver. Pay attention to the following methods:

- The constructor. WiringPi is initiated here. This gives it the mappings to the pins on the Raspberry Pi board.
- `maker()`. Every plugin needs the maker function. Without it the driver will not exist in memory at all.

- `getCapabilities()`. Here we define a property `led_state` which is a boolean. Changing this boolean value will operate the LED.

- `setDeviceProperty()`. In this method we recognize the given property name and set the corresponding state of the LED.

For pin value modifications there is a helper method called `setGpioPort()`. This will use Wiring Pi methods to set the pin as output and then `digitalWrite` the boolean value to the pin to set it high or low.

```
#include "RaspberryGpioLedDriver.h"
#include <stdio.h>
#include <wiringPi.h>

#define PINNUMBER 7    // aka BCM_GPIO pin 4


RaspberryGpioLedDriver::RaspberryGpioLedDriver() : CustomHwDriverPlugin()
{
 wiringPiSetup();
}

const QList<InterfaceCapability> RaspberryGpioLedDriver::getCapabilities()
{
 QList<InterfaceCapability> l;

 // Capabilities of the driver
 InterfaceCapability ledEntry;
 ledEntry.type = InterfaceCapabilityType::CapabilityTypeProperty;
 ledEntry.canRead = true;
 ledEntry.canWrite = true;
 ledEntry.canStorePermanently = false;
 ledEntry.description = "On/off control for an LED";
 ledEntry.name = "led_state";
 ledEntry.datatype = QVariant::Bool;
 l << ledEntry;

 return QList<InterfaceCapability>(l);
}

bool RaspberryGpioLedDriver::write(QVariant address, QByteArray data)
{
 setGpioPort(address.toInt(), data.at(1));
 return true;
}

int RaspberryGpioLedDriver::setDeviceProperty(QVariant propertyId, QVariant propertyValue)
{
 if (propertyId.toString() == "led_state") {
  switchLed(propertyValue.toBool());
  emit devicePropertyReceived(propertyId, propertyValue);
 }
 return 0;
}

void RaspberryGpioLedDriver::switchLed(bool direction)
{
 if (direction)
  setGpioPort(PINNUMBER, 1);  // ON (high)
 else
  setGpioPort(PINNUMBER, 0);  // OFF (low)
}

void RaspberryGpioLedDriver::setGpioPort(int portNumber, int portValue)
{
 pinMode(portNumber, OUTPUT);
 digitalWrite(PINNUMBER, portValue);
}

extern "C" {

__attribute__ ((visibility ("default")))
plugin* maker() {
 return new RaspberryGpioLedDriver ;
}

} // extern c
```

Once you have the driver source code on your Raspberry Pi, compile it:

```
qmake
make
```

Now you have a sample driver. Copy it to the plugins directory of Asema IoT Edge installation and restart Asema IoT Edge software.

To do LED controls, you will need to attach an LED onto your Raspberry Pi board. In this case it should reside between ground and PIN 4 (probably with a suitable resistor to protect it).

To control the LED, create an object into Asema IoT Edge. In the Asema IoT Edge Web admin interface, go to Objects > Hardware > Custom. Add a new object and chooce your shiny new driver as its driver.

Next, when you open the control dashboard of this object (in Web admin interface, go to Objects > Control and monitor and select your object), you can now add a toggler to the dashboard and it will change the LED value.