English

Asema IoT Central
Plugin Interface 1.0

Table of Contents

1. Introduction 1
    1.1. The purpose of plugins 1
    1.2. Types of plugins 1
2. Compiling and loading plugins 2
    2.1. Using the developer library package 2
    2.2. Compiling a plugin 2
    2.3. Mandatory methods for each plugin 3
    2.4. Placing a plugin in the system 3
3. Object handling plugins 4
4. Hardware driver plugins 6
5. Positioning plugins 13
6. Routing plugins 14
7. Smart API plugins 15

1. Introduction

1.1. The purpose of plugins

Plugins are a way to extend the functionality of Asema IoT Central. In practice they are dynamically loaded libraries, programmed in C++, with a certain API that makes it possible for the system to call them. While many of the Asema IoT system features can be either configured or scripted with JavaScript, there are still many use cases that either need more specific data processing or must offer more processing power than a script can offer. This is where plugins come along.

An example of a very common and useful plugin is a hardware driver. Custom hardware typically requires some access to system services, serial buses and similar features of the host. Scripting or high level programming seldom has access to them due to sandboxing. This is why hardcore, C++ level code is the only proper and possible way to access them. With a plugin this access can be achieved properly.

Inside Asema IoT Central and Asema IoT Edge software, plugins are handled by a plugin manager that looks for them in a specific place in the system installation (more about this later). Plugins are loaded dynamically: the load happens only after the first request to plugins services. Plugins linked to objects that load during startup, naturally load and activate when the objects load i.e. at system startup.

There are several types of plugins and each plugin type has a certain API. All plugins inherit the base classes that give the plugin system basic plugin info such as its name and type. The rest is type-specific. This manual describes how each plugin type can be implemented.

1.2. Types of plugins

Currently, Asema IoT Central supports the following plugin types:

- **Object handling plugins**. Object handling plugins perform custom parsing and serializing of input and output data between an external data source and an object.
- **Hardware driver plugins**. Hardware driver plugins connect to hardware and perform conversions between object data and the proprietary data format used by the hardware. This hardware usually resides on the same host machine but can also operate over a network.
- **Positioning plugins**. Positioning plugins provide positional (latitude, longitude, altitude) data. A typical positioning plugin would be a driver for a custom GPS chip.
- **Routing plugins**. Routing plugins calculate routes between points. They respond with a data structure containing the segments and points to a destination.
- **Smart API plugins**. Smart API plugins are used to process customized data structures made with the Smart API library.

Asema IoT Edge supports object handling plugins, hardware driver plugins, and positioning plugins

Chapters later in this manual show in detail how each plugin type works and can be programmed.

2. Compiling and loading plugins

2.1. Using the developer library package

The Asema IoT developer library is a C++ library against which you can compile code compatible with the Asema IoT binaries. You will need these headers to compile your code.

Download the package from https://iot.asema.com/iotc/downloads.html, unpack it and follow the compiling instructions below.

2.2. Compiling a plugin

Asema IoT has been developed using the Qt programming library and tools. To be able to compile plugins, you will need the same. So to start with, go to https://www.qt.io and download the necessary packages. Then install Qt an the necessary tools. What you will eventually need is at minimum `qmake` and `make`. Qt's installation documentation will show you how to get these.

Once you have Qt installed, you will need a project (.pro) file. The project file is used by `qmake` as a basis for generating a makefile for your plugin. You can make the project file either with the Qt Creator tool that comes with the Qt library or by hand. When you run `qmake` on this file, you get a makefile which is then the input for `make`. So the compile process for a plugin is in its simplest form as follows:

```
qmake [name of project file]
make
```

The project file will contain includes of the required Qt libraries, paths to other library files to include, compile flags, and a list of the plugin files itself. Qt's documentation describes in detail the different parts and their meaning. The actual content of a particular project file will naturally depend of what your plugin needs, but let's start with a simple example. The following project file is for a plugin called "nop" (No OPeration) that simply loads but performs no other action.

```
QT += core qml network sql

TARGET = nop
TEMPLATE = lib

DEVEL_LIBS_HEADERS = "./developerlibs/1.0/"
QMAKE_CXXFLAGS += -std=gnu++11

INCLUDEPATH += $${DEVEL_LIBS_HEADERS}

SOURCES += NopPlugin.cpp
HEADERS += $${DEVEL_LIBS_HEADERS}/plugin/plugin.h \
 $${DEVEL_LIBS_HEADERS}/plugin/systemplugin.h \
 $${DEVEL_LIBS_HEADERS}/plugin/processingplugin.h \
 $${DEVEL_LIBS_HEADERS}/network/handlers.h \
 NopPlugin.h

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

Important
In the project (.pro) file, remember to include references to **all** the classes in the plugin inheritance path relevant to your plugin. Failure to do so will result in missing symbols during runtime and failure to load the plugin even though it compiles without problems.

2.3. Mandatory methods for each plugin

Now that we know how to compile a plugin, let's take a look at the plugin structure in more detail. Here is the code for the nop (i.e. no operation) plugin that is capable of compiling and loading but does nothing else.

```
#ifndef __NOPPLUGIN_H
#define __NOPPLUGIN_H

#include "plugin/systemplugin.h"

class NopPlugin  : public SystemPlugin
{
 Q_OBJECT

public:
 NopPlugin() : SystemPlugin() {}

 void* process(int, QVariant);
 const char* getName() { return "NopPlugin"; }
};

#endif // __NOPPLUGIN_H
```

```
#include "NopPlugin.h"

extern "C" {

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

} // extern c

void* NopPlugin::process(int callId, QVariant dataObject)
{
 Q_UNUSED(dataObject)
 emit processed(callId, QVariant());
 return NULL;
}
```

A plugin has two mandatory methods you must implement: `getName()` and `getType()`. This sample plugin inherits the `SystemPlugin` which already implements `getType()` so the only thing left is `get-Name()` which simply returns the name of the plugin. A `SystemPlugin` has another mandatory virtual method `process` but more on that later.

Another important mandatory method of a plugin is `maker()`. This method is the one that creates and instance of this particular plugin. This method must be implemented as shown and with that name. It is called by the plugin handling logic of Asema IoT to actually get an instance of the plugin it needs to run.

2.4. Placing a plugin in the system

Once you have a compiled plugin (stored in a .dll or .so), copy this file into Asema IoT plugins directory. This can be found in the data directory of Asema IoT. If you don't know where that is, open the web admin interface of your Asema IoT installation and go to System > Paths and identifiers. At the bottom of the listing there is a section that shows paths to all plugin directories. Copy the .dll or .so files there. Then restart Asema IoT.

Important
When you develop a plugin, remember to follow the system output log carefully. If your plugin is missing symbols or fails to load due to some other reason, you will see error messages in the log.

3. Object handling plugins

Object handling plugins are meant as data conversion tools between Asema IoT's object model and some proprietary data format . They perform two basic functions: (1) parsing i.e. taking some data and extracting values from it, and (2) serializing i.e. transforming object data into the proprietary data. The data is inputted and outputted as byte arrays so the proprietary data can contain anything from just an unstructured byte stream to some document.

In addition to the parsing and serialization methods, object handling plugins have two helper methods for handling object metainformation. These methods should fill in the fields of a data entry form. When a user enters the metadata editing of an object in the web admin interface (or some other interface where metadata editing can be done), the system will fetch the metadata fields supported by the object from the backend. The plugin is then asked to fill in the fields. If there is a plugin associated with the object, the custom metadata and property fields are set according to the spec given by the plugin and displayed to the user.

The fields in the form have a label, a datatype, a flag indicating whether filling the field is mandatory, and some default value. Let's look at an example of creating the fields of such a form.

```
void MySampleObjectPlugin::fillMetaFields(QVariantMap *metadata)
{
 if (!metadata->contains("entry_fee")) {
  QVariantMap item;
  item.insert("label", "Entry fee");
  item.insert("type", "float");
  item.insert("default_value", "10");
  item.insert("mandatory", false);
  metadata->insert("entry_fee", item);
 }

 if (!metadata->contains("mandatory_age")) {
  QVariantMap item;
  item.insert("label", "Mandatory age");
  item.insert("type", "integer");
  item.insert("default_value", false);
  item.insert("mandatory", false);
  metadata->insert("mandatory_age", item);
 }
}
```

An object handling plugin has the following interface that needs to be implemented:

`QByteArray serialize()` (*data*);

`QVariantMap` *data* `Data to serialize.;`
Transform the data in a key-value map of property values (data) into some format understood by the recipient and represented in bytes that can be sent over a socket to the recipient.

`QByteArray serialize()` (*data*, *errors*);

`QVariantMap` *data* `Data to serialize.;`
`QList<QString>*` *errors* `Pointer to error structure.;`
Transform the data in a key-value map of property values (data) into some format understood by the recipient and represented in bytes that can be sent over a socket to the recipient. If errors occur, write the error messages as a list to the errors list.

`QByteArray serialize()` (*data*, *obj*, *schema*, *errors*);

`QVariantMap` *data* `Data to serialize.;`
`CoreObject*` *obj* `Object to serialize.;`
`QVariantMap` *schema* `Schema to use in serialization.;`
`QList<QString>*` *errors* `Pointer to error structure.;`
Transform the data in an object described by the schema. If the object does not have a value, take the value from a key-value map (data). If errors occur, write the error messages as a list to the errors list.

`QVariantList parse()` `(`*`data`*`);`

`QByteArray` *`data`*`;`
Parse data represented by an array of bytes and transform it into a property list (maps of key-value pairs).

`QVariantList parse()` `(`*`data`*`, `*`errors`*`);`

`QByteArray` *`data`*`;`
`QList<QString>*` *`errors`* `Pointer to error structure.;`
Parse data represented by an array of bytes and transform it into a property list (maps of key-value pairs). If errors occur, write the error messages as a list to the errors list.

`void* process(int callId, QVariant userData, CoreObject* object)` `(`*`callId`*`, `*`user-Data`*`, `*`obj`*`);`

`int` *`callId`*`;`
`QVariant` *`userData`*`;`
`CoreObject*` *`obj`* `Object to serialize.;`

`void fillMetaFields()` `(`*`metaForm`*`);`

`QVariantMap*` *`metaForm`*`;`
Write a form structure for inputting metadata fields of some device this plugin is made to represent.

`void fillPropertyFields()` `(`*`propertyForm`*`);`

`QVariantMap*` *`propertyForm`*`;`
Write a form structure for inputting custom property fields of some device this plugin is made to represent.

4. Hardware driver plugins

Hardware driver plugins implement the required driver interface. This interface spans a number of methods from initiating the driver to setting various parameters to it at runtime. That said, while the interface is extensive, not all of it has to be implemented. The depth of implementation depends on your driver and its needs. For instance, if it cannot be parametrisized in any way, then the methods related to getting and setting parameters can simply be left empty. Similarly, if your hardware does not support two-way communication, methods related to setting some values are unnecessary.

If your hardware on the other hand does have the communication possibilities and processing power, it is highly recommended to implement most of the debugging and monitoring methods as this makes life a lot easier when problems with devices on the field occur.

The following code is an example of a recommended driver structure. Let's start with the header file and analyze it:

```
#ifndef __MYCUSTOMHWDRIVERPLUGIN_H
#define __MYCUSTOMHWDRIVERPLUGIN_H

#include "plugin/customhwdriverplugin.h"
#include "objects/coreobject.h"

// As we are not making a "real" driver in the example
// a timer is used to simulate some periodic automatic
// functionality
#include <QTimer>

class MyCustomHwDriverPlugin  : public CustomHwDriverPlugin
{
 Q_OBJECT

public:
 MyCustomHwDriverPlugin ();

 // This method is mandatory
 const char* getName() { return "TimerDriverPlugin"; }

 // Most drivers will want some initialization routine
 bool initDriver();

 // Connection methods are needed in a connected protocol
 void discoverAndConnectToDevice();
 void connectToDevice();
 void disconnectFromDevice();

 // These are mandatory for property handling
 QVariant getDeviceProperty(QVariant propertyId);
 int requestDeviceProperty(QVariant propertyId);
 int setDeviceProperty(QVariant propertyId, QVariant propertyValue);

 // This method is mandatory if you want proper admin user interfaces and
 // support sharing for the device
 const QList<InterfaceCapability> getCapabilities();

 // Implementing these four is highly recommended for error
 // handling purposes
 int selfDiagnoseDevice();
 int locateDevice();
 int pingDevice();
 int restartDevice();

 // These methods usually come in handy when debugging connections
 QString getAddress();
 int setDeviceClock(quint64 mseconds_from_epoch);
 int requestDeviceConnectionState();
 int requestDeviceStatus();
 int requestDeviceConnectionQuality();
 int requestDeviceFirmwareVersion();
 int requestDeviceClock();


 int unpairDevice() { return 0; }
 bool write(QVariant address, QByteArray data) { Q_UNUSED(address); Q_UNUSED(data); return false; }
 QByteArray read(QVariant address) { Q_UNUSED(address); return QByteArray(); }
 int calibrateDevice(int calibrationType) { Q_UNUSED(calibrationType); return 0; }
 int setDeviceCalibrationParameters(QByteArray calibrationData) { Q_UNUSED(calibrationData); return 0; }

 QVariant getDeviceParameter(QVariant parameterId) { Q_UNUSED(parameterId); return QVariant(); }
 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; }
 const QMap<QString, QString> getParametersOfDevice() { return QMap<QString, QString>(); }
 int requestParametersFromDevice() { return 0; }
 int setParametersToDevice(QMap<QString, QString> attributeData) { Q_UNUSED(attributeData); return 0; }

 QVariant getDeviceState(QVariant stateId) { Q_UNUSED(stateId); return QVariant(); }
 int requestDeviceState(QVariant stateId) { Q_UNUSED(stateId); 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 requestDeviceValue() { return 0; }
 int setDeviceValue(QVariant value) { Q_UNUSED(value); return 0; }

signals:
 // As of Asema IoT, these two signals must be explicitly defined in the header of the plugin
 // the rest are inherited.
 void deviceClockReceived(quint64 clockReading);
 void deviceLocation(QGeoPositionInfo location);

private slots:
 void onTimeout();

private:
 QTimer mTimer;
 double mDistanceValue;
 double mAccelerationValue;
 quint64 mCurrentClock; // There is no "clock" on this simulated sample, just store into a local variable
};

#endif // __MYCUSTOMHWDRIVERPLUGIN_H
```

The header is for a plugin stub that does not actually connect to live hardware but shows how to react to calls from the system and signal back the results. To have something happening in a simulated manner, the driver runs a timer that ticks once per second, advancing inner values and signaling those forward. Asema IoT system will react to those signals as if they originate from real hardware.

Noticeable methods implemented in the class include:

- `getName()`. This name is displayed in various places in the UI and is mandatory for the plugins management. Keep the name unique.
- `connectToDevice()`. Called by the system when the object loads and when you explicitly invoke a connect. If you have any kind of device with a connected protocol, this is a must.
- `disconnectFromDevice()`. The opposite of connect. Required if you want a restart / boot.
- `getDeviceProperty()`and `setDeviceProperty()`. Property getter and setter. These methods are the core link to the object property system so do implement these abstractions.
- `getCapabilities()`. The basis of schemas. This is basically a discovery method that lists the names of the properties your driver abstracts. Used by UI's, sharing, APIs, and many more.

So how do we go about implementing this class? Let's look at some sample implementation of the main methods to use:

```
extern "C" {

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

This first one is an important one, the maker function. You need to have one as this is the method that creates an instance of the class and stores it into the plugin system memory. Without it your plugin never loads. The structure is very simple: you have one method, always called `maker()` which returns one instance of your class. That's it.

```
MyCustomHwDriverPlugin::MyCustomHwDriverPlugin () :
 CustomHwDriverPlugin(),
 mDistanceValue(0),
 mAccelerationValue(0)
{
 mCurrentClock = QDateTime::currentDateTime().toMSecsSinceEpoch();
 connect(&mTimer, SIGNAL(timeout()), this, SLOT(onTimeout()));
 mTimer.start(1000);
}
```

Next, the constructor. Obviously you need to put here the initializations your class needs. This sample plugin simply sets internal variables to zero and starts a timer. Remember to call the constructor of the parent class, too!

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

 // Sample capabilities for the driver
 InterfaceCapability distanceEntry;
 distanceEntry.type = InterfaceCapabilityType::CapabilityTypeProperty;
 distanceEntry.canRead = true;
 distanceEntry.canWrite = true;
 distanceEntry.canStorePermanently = false;
 distanceEntry.description = "Distance traveled";
 distanceEntry.name = "distance";
 distanceEntry.datatype = QVariant::Double;
 l << distanceEntry;

 InterfaceCapability accelerationEntry;
 accelerationEntry.type = InterfaceCapabilityType::CapabilityTypeProperty;
 accelerationEntry.canRead = true;
 accelerationEntry.canWrite = true;
 accelerationEntry.canStorePermanently = true;
 accelerationEntry.description = "Acceleration of the vahicle";
 accelerationEntry.name = "acceleration";
 accelerationEntry.datatype = QVariant::Double;
 l << accelerationEntry;

 return l;
}
```

The `getCapabilities()` method is the plugin's way to expose the property handling it supports. This is done with a list of objects of the `InterfaceCapability` type. Each `InterfaceCapability` tells what capabilities the interface where the driver is attached to can support, including the name of the property supported, whether it is only a read or a read-write, and what the datatype to feed to it should be. The name given to the `InterfaceCapability` is the property name. This name is the string that is fed back by the system to the getters and setters to request and set values. So match entries here with `propertyId` values of `getDeviceProperty()` and `setDeviceProperty()` methods.

```
void MyCustomHwDriverPlugin::connectToDevice()
{
 connected = true;
 emit deviceOnline();
}
```

The way you implement a connection to the device is of course up to the implementation, but once done, the "connected" flag should be set to true and a successful connection signal should be sent (note that this operation can be asynchronous).

```
void MyCustomHwDriverPlugin::disconnectFromDevice()
{
 connected = false;
 emit deviceOffline();
 }
```

The same thing with disconnect. Once done, flag off and emit a signal.

```
int MyCustomHwDriverPlugin::pingDevice()
{
 if (connected)
  emit deviceHeartBeat();
 return 0;
}
```

Here's an example implementation of pinging, i.e. trying to connect to the device for a simple connection test. The signal to be sent back is `deviceHeartBeat()`. The heartbeat is simply a signal sent periodically to say that the system is alive. Note that you can send heartbeats regularly even when there is no explicit ping call.

```
QVariant MyCustomHwDriverPlugin::getDeviceProperty(QVariant propertyId)
{
 if (propertyId.toString() == "acceleration") return mAccelerationValue;
 else if (propertyId.toString() == "distance") return mDistanceValue;
 return QVariant();
}

int MyCustomHwDriverPlugin::setDeviceProperty(QVariant propertyId, QVariant propertyValue)
{
 if (propertyId.toString() == "acceleration") mAccelerationValue = propertyValue.toDouble();
 else if (propertyId.toString() == "distance") mDistanceValue = propertyValue.toDouble();
 emit devicePropertyWritten(propertyId, propertyValue);
 return 0;
}
```

Here's the getter and setter sample. What exactly is done in your hardware device with each property is up to your implementation but the core idea is this: you will get a propertyId that will match the name of the interface capability exposed in the `getCapabilities()` method. Note that the get and set functionality in your system may be asyncronous (and often is if youre using a wireless connection or a slow link). `getDeviceProperty` should however always return some known value of the property, then emit an updated value later. There is a separate method `int requestDeviceProperty(QVariant propertyId)` which is an explicitly asyncronous request for a value. This method does not return the property value directly but should instead use the `void devicePropertyReceived(QVariant propertyId, QVariant propertyValye)` signal to send the value once obtained from hardware.

```
int MyCustomHwDriverPlugin::requestDeviceClock()
{
 emit deviceClockReceived(mCurrentClock);
 return 0;
}
```

Finally, as sample of a typical debugging/ management function. Getting the clock. All management functions work with the same asynchronous logic. You get a call to the method, do some processing and once done, signal the result. You can find the various signals that can be sent in `hardwarecontroller.h` in the developer library.

`bool initDriver()`

Driver initialization routine. Called when the plugin loads (as opposed to when it is created, i.e. the constructor). Place the initializations you need the driver to perform here.

`void discoverAndConnectToDevice()`

Run a service discovery (as in e.g. Bluetooth) and a connect on a device. Usually also does pairing.

`void connectToDevice()`

Do connect to a hardware device (without discovery).

`void disconnectFromDevice()`

Close connection to the hardware.

`int unpairDevice()`

Run an unpair operation (e.g. in Bluetooth) on the device.

`bool write(QVariant address, QByteArray data)`

Actions to perform when the system asks to write a set of raw bytes to a given address.

`QByteArray read(QVariant address)`

Actions to perform when the system asks to read a set of raw bytes from a given address.

`int calibrateDevice(int calibrationType)`

Run a calibration operation. Could be done in a test bed for instance to make sure sensor settings are accurate.

`int selfDiagnoseDevice()`

Run a self-diagnosis sequence on the hardware.

`int locateDevice()`

Find the device. Could for instance blink an LED or make a sound to recognize a particular device if there are many in the vicinity. May optionally return the coordinates of the device with a signal.

`int pingDevice()`

Test device connection. If successful, should return a heartbeat signal as a result.

`int restartDevice()`

Run a restart/boot on the hardware.

`int setDeviceCalibrationParameters(QByteArray calibrationData)`

Input a set of calibration values into the hardware for instance at the end of a test bed run to make sure e.g. sensoring hardware measures correctly.

`int setDeviceClock(quint64 milliSecondsFromEpoch)`

If the device has a clock, set it to this value.

`int requestDeviceConnectionState()`

Run diagnosis of the connection (e.g. online / offline) and signal back the status.

`int requestDeviceStatus()`

Run diagnosis of the hardware device (e.g. OK, misconfigured, error detected) and signal back the status.

`int requestDeviceConnectionQuality()`

Run diagnosis of the connection quality (e.g. bit loss, signal strength) and signal back the status.

`int requestDeviceFirmwareVersion()`

Request the version info of the device, signal back the version number as a string.

`int requestDeviceClock()`

Request the current clock value, signal back the clock in milliseconds from epoch.

`QVariant getDeviceProperty(QVariant propertyId)`

Return the last known value of a property identified by propertyId.

`int requestDeviceProperty(QVariant propertyId)`

Run a request of the propery value on the hardware and signal the fetched value once done.

`int setDeviceProperty(QVariant propertyId, QVariant propertyValue)`

Set the property value (or whatever the property abstracts) on the device.

`QVariant getDeviceParameter(QVariant parameterId)`

Get the latest known value of the named configuration parameter of the device.

`int requestDeviceParameter(QVariant parameterId)`

Fetch and update the value of the named configuration parameter from the device and signal the result when done.

`int setDeviceParameter(QVariant parameterId, QVariant parameterValue)`

Set the configuration parameter value on the device.

`int setDevicePermanentParameter(QVariant parameterId, QVariant parameterValue)`

Set the configuration parameter value and store it permanently (e.g. on flash memory).

`const QMap<QString, QString> getParametersOfDevice()`

Return a map of the last known values and names of configuration parameters of a device.

`int requestParametersFromDevice()`

Request all configuration parameters of a device. Signal a map of key-value pairs once done.

`int setParametersToDevice(QMap<QString, QString> attributeData)`

Set values of configuration parameters from a map of key-value pairs.

`QVariant getDeviceState(QVariant stateId)`

Get the last known value of a state variable on the device.

`int requestDeviceState(QVariant stateId)`

Request the value of a state variable of the device. Signal the value back once received.

`int setDeviceState(QVariant stateId, QVariant stateValue)`

Set the state variable value on the device.

`int setDevicePermanentState(QVariant stateId, QVariant stateValue)`

Set the state variable value on the device and store it permanently (e.g. on flash).

`int requestDeviceValue()`

Request the value that the device represents if it is a single-use device (e.g. a temperature sensor represents temperature).

`int setDeviceValue(QVariant value)`

Set the value that the device represents if it is a single-use device (e.g. a relay represents on/off state).

`const QList<InterfaceCapability> getCapabilities()`

List the properties this driver supports with read/write/store capabilities.

`QString getAddress()`

Get the address of the device if applicable in the format commonly used for a communication protocol in case (for instance a MAC address).

5. Positioning plugins

A positioning plugin has only one method to implement, the `positionUpdated()` signal. This signal should be sent whenever a new position has been calculated. The parameter of the `positionUpdated` signal is a `QGeoPositionInfo` which basically contains the coordinates of the updated position.

A position can also be given in relation to something else or with the help of something else. This is the case with, for example, indoor positioning where the positioning is calculated in relation to beacon stations. This "something else" is represented by other objects in Asema IoT. For this purpose, a positioning plugin can be given these references as a list with the `setReferenceObjects()` method. What the plugin actually does with the references is up to the code of the plugin. When references are obtained by the plugin, the plugin gets access to the full API of the object and can for instance call the `latitude()` and `longitude()` methods of the object to obtain its position. It can also connect to any of the signals the object sends and in this way calculate position whenever something in the reference object changes.

Note
The positioning plugin is for the **system**that runs Asema IoT software i.e. some computer hardware or embedded device that runs Asema IoT Central or Asema IoT Edge. For positioning an object, you would use the `setLatitude()`, `setLongitude()`, and `setAltitude()`methods of the `CoreObject`. Access to these methods in turn should be implemented in a hardware driver for that object, not a positioning plugin.

Note
Only one positioning plugin can be in use at a time. To choose which plugin to load as the positioning plugin of the system, you need to open the Web admin interface and go to System > Location. The section "Available positioning plugins" will list the plugins detected by the system. Here you can choose which on is active.

`void positionUpdated(QGeoPositionInfo pos)`

Signal to send when a new position is acquired. Signal data should contain at minimum the new latitude and longitude.

`void setReferenceObjects(QList<CoreObject*> ro)`

Feed into the plugin a list of objects that the plugin may use as reference points for positioning (e.g. positions of a set of beacons used to trilaterate a position).

6. Routing plugins

Routing plugins calculate routes (i.e. a set of waypoints to travel through) based on input given to them. The way the plugin makes the calculation depends on the way the plugin is programmed. As the majority of mapping applications work on geospatial data stored on a server, the most common plugin operation is to calculate the route by sending a routing request to a server. However, if map data is locally stored, the plugin may do the calculation locally as well.

The input to a routing plugin is **not** structured. There are no structs or classes that would define what goes into the input, except that it needs to be a map. This is for a purpose: many routing algorithms require additional data input from the users or some custom parameters to work with. Otherwise there really would not be a particular need for custom routing, everyone would be just fine with standard routing methods already available. This is why plugin input is just a map. Applications (Web applications, Screenlets) may place whatever data is needed and is compatible with the plugin into it. Asema IoT system does not tamper with the data as it flows to the plugin, it is simply passed on "as is".

That said, a common structure for routing plugin input would be a map that contains a list of waypoints the route needs to pass through. Each waypoint in turn is a map that contains a latitude and a longitude. For example, expressed in JSON:

```
{
 "waypoints": [
  { "latitude": 64.5678, "longitude": 23.4555 },
  { "latitude": 64.6789, "longitude": 23.5666 },
  { "latitude": 64.7891, "longitude": 23.6777 }
 ]
}
```

A routing plugin is a processing plugin, so once the route is calculated, it should emit the calculation result with the `processed()` signal. The signal comprises a `callId` and the result of the routing. The same applies to the plugin output as to the input: there is no prespecified format for it. Custom routing algorithms may want to deliver some fancy results to justify their existence and their output cannot therefore be limited. Again, the output is probably a list of route points or most likely a list of alternative routes (list of lists of route points). Like so:

```
{
 "routes": [
  {
   "name": "Road E40",
   "distance": 310
   "routePoints": [
    { "latitude": 64.5678, "longitude": 23.4555 },
    { "latitude": 64.8900, "longitude": 23.0000 },
    { "latitude": 64.8901, "longitude": 23.0001 },
    { "latitude": 64.7891, "longitude": 23.6777 }
   ]
  },
  {
   "name": "Road E501",
   "distance": 362
   "routePoints": [
    { "latitude": 64.5678, "longitude": 23.4555 },
    { "latitude": 64.9001, "longitude": 23.0000 },
    { "latitude": 64.9002, "longitude": 23.0001 },
    { "latitude": 64.9003, "longitude": 23.0002 },
    { "latitude": 64.7891, "longitude": 23.6777 }
   ]
  }
 ]
}
```

`void calculateRoute(int callId, QVariantMap routeInput)`

Invoke route calculation with parameters represented by a map of input values (typically waypoints).

7. Smart API plugins

Smart API plugins are for custom processing of Smart API data. Asema IoT processes incoming Smart API requests with its standard processor that assumes the structure defined by the standard. But there may be various applications that need a custom format with additional fields or a new ontology altogether. The purpose of these plugins is to enable the processing of such requests and mapping the values of processing into Asema IoT objects.

There are two things a Smart API plugin can do: act as a processor for a full `Activity`, i.e. a request processor for a service (which is represented and identified by an `Activity`) in Smart API or handle specific objects.

Activity handling takes place by a call to the `start()` method which takes a pointer to the activity as its arguments. Usually this is enough. The plugin should in this case look for the contents of the input and proceed accordingly. As the Smart API plugin is a `ProcessingPlugin`, it should present the processing results by emitting a `processed()` signal. If the processing takes long, the plugin may also receive a call to the `end()` method that asks the plugin to cancel the processing.

Object processing takes place with the `read()` and `write()` methods. Both get as arguments the Smart API side of processing (`Entity`) and the system side of processing (`CoreObject`). In a write, the plugin should take values from the `Entity` and place them into the `CoreObject`. In a read, the operation should be exactly the opposite.

`void start(Activity* activity)`

Start processing some Activity in an asynchronous manner. Once the Activity has been handled, should send a processed() signal to notify that it is done.

`void end()`

End processing an Activity before the processed() signal has been sent.

`void write(Entity* source, CoreObject* target, TemporalContext* time)`

Write values represented by a Smart API Entity into an Asema IoT CoreObject target (in whichever manner the application in case wants to do that). If a range of values over time should be written, the timespan can be found in the TemporalContext time.

`void read(Entity* target, CoreObject* source, TemporalContext* time)`

Read values from an Asema IoT CoreObject source and write them to a Smart API Entity target (in whichever manner the application in case wants to do that). If a range of values over time should be written, the timespan can be found in the TemporalContext time.

Asema Electronics Ltd
Copyright © 2011-2019