DLT Viewer Plugins Programming Guide

Alexander Wenzel

May 18, 2018

DLT Viewer Plugins Programming Guide
by Alexander Wenzel

Contents


GENIVI®

Chapter 1

Overview

The DLT Viewer's functionality can be extended by plugins. There are currently four different types of plugins:

Decoder Plugins Interprets DLT messages and translates them into human readable data. The decoded message content is also used to filter the messages.

Viewer Plugins These plugins create UI widgets which can display additional graphical information about DLT messages. They can be used with individual selected messages or whole log streams and DLT files.

Control Plugins These plugins are able to control the message table of the DLT Viewer and to communicate with target applications.

Command Plugins These plugins can read command line arguments passed to the DLT Viewer.

Each plugin can be a mixture of the types listed above, depending on which interfaces the plugins implement.

Chapter 2

Interfaces

The plugins can implement several plugin interfaces:

QDLTPluginInterface This is the standard DLT Viewer plugin interface. This interface **must** be inherited by each DLT Viewer plugin.

QDLTPluginDecoderInterface This is an extended DLT Viewer plugin interface. This interface must be used by decoder plugins. DLT messages are passed to and can be decoded by the plugin. The plugin decides on its own which messages to decode.

QDltPluginViewerInterface This is an extended DLT Viewer plugin interface. This interface must be used by viewer plugins. The viewer plugin gets full access to the loaded DLT file. Viewer plugins provide a UI widget to display additional information or other graphical content.

QDltPluginControlInterface This is an extended DLT Viewer plugin interface. This interface must be used by control plugins. Control plugins get informed about the available connections to DLT daemons. They can send control requests to and receive responses from target applications. The plugin can also "jump" inside the message table in order to guide the user's focus to specific messages.

QDltPluginCommandInterface This is an extended DLT Viewer plugin interface. This interface must be used by command plugins. The command plugin interface can be used to receive commands passed from the UI or command line.

The QDLTPluginInterface must be implemented by every plugin. The other plugin interfaces are optional.

Chapter 3

A Basic Plugin

The following code samples show the minimal code needed for a DLT Viewer plugin. (You can find these files in the DLT Viewer project.)

dummyplugin.h

```
#ifndef DUMMYPLUGIN_H
#define DUMMYPLUGIN_H

#include <QObject>
#include "plugininterface.h"

#define DUMMY_PLUGIN_VERSION "1.0.0"

class DummyPlugin : public QObject, QDLTPluginInterface
{
    Q_OBJECT
    Q_INTERFACES(QDLTPluginInterface)

public:
    DummyPlugin();
    ~DummyPlugin();

    /* QDLTPluginInterface interface */
    QString name();
    QString pluginVersion();
    QString pluginInterfaceVersion();
    QString description();
    QString error();
    bool loadConfig(QString filename);
    bool saveConfig(QString filename);
    QStringList infoConfig();
};

#endif // DUMMYPLUGIN_H
```

dummyplugin.c

```
#include <QtGui>

#include "dummydecoderplugin.h"

DummyPlugin::DummyPlugin()
{}

DummyPlugin::~DummyPlugin()
{}

QString DummyPlugin::name()
{
    return QString("Dummy Plugin");
```

```
}

QString DummyPlugin::pluginVersion()
{
    return DUMMY_PLUGIN_VERSION;
}

QString DummyPlugin::pluginInterfaceVersion()
{
    return PLUGIN_INTERFACE_VERSION;
}

QString DummyPlugin::description()
{
    return QString();
}

QString DummyPlugin::error()
{
    return QString();
}

bool DummyPlugin::loadConfig(QString /* filename */ )
{
    return true;
}

bool DummyPlugin::saveConfig(QString /* filename */)
{
    return true;
}

QStringList DummyPlugin::infoConfig()
{
    return QStringList();
}

#ifndef QT5
Q_EXPORT_PLUGIN2(dummyplugin, DummyPlugin);
#endif
```

dummyplugin.pro

```
# include global settings for all DLT Viewer Plugins
include( ../plugin.pri )

# target name
TARGET = $$qtLibraryTarget(dummyplugin)

# plugin header files
HEADERS += \
    dummyplugin.h

# plugin source files
SOURCES += \
    dummyplugin.cpp

# plugin forms
FORMS +=
```

Additional interfaces and thus additional functionality can be added by deriving the plugin class from any of the other interfaces. Remember that the QDLTPluginInterface must still be implemented, even if you add one or more of the optional plugin interfaces.

```
class DummyDecoderPlugin : public QObject, QDLTPluginInterface,  ↩
    QDLTPluginDecoderInterface
```

You also have to declare the interfaces you use with the `Q_INTERFACES` macro in the plugin's header file.

```
   Q_INTERFACES(QDLTDecoderPluginInterface)

public:
    /* QDltPluginDecoderInterface */
    bool isMsg(QDltMsg &msg, int triggeredByUser);
    bool decodeMsg(QDltMsg &msg, int triggeredByUser);
```

All interface functions are *pure virtual functions*, so they must be defined in the plugin's `.cpp` file.

If you also implemented a widget for viewer plugins, add a form to the plugin project and return an instance of the widget in the `initViewer()` function.

IMPORTANT

If you return a UI widget in the `initViewer()` function, this widget will be owned by the Qt framework, which will handle the widget's destruction. Do not try to delete this widget on your own!

Chapter 4

QDLTPluginInterface description

This is the standard DLT Viewer Plugin Interface. This interface must be inherited by each DLT Viewer plugin. Most of the functions in this interface are called when loading the plugin the first time.

4.1 Function `QString name()`

The plugin must provide a name. This name is shown to the user in the project configuration.

4.2 Function `QString pluginVersion()`

The plugin has to return a version number in the format `X.Y.Z`.

- *X* should count up in case of really heavy changes (API changes or purpose changes)
- *Y* should count up when the module is reworked internally, functions are added etc
- *Z* should count up whenever a bug is fixed

Recommendation: `#define <plugin name>_PLUGIN_VERSION "X.Y.Z"` in your plugin header file.

4.3 Function `QString pluginInterfaceVersion()`

The plugin has to return a version number of the used plugin interface. The plugin interface provides the `PLUGIN_INTERFACE_VERSION` definition for this purpose.

4.4 Function `QString description()`

Currently not used by DLT Viewer.

4.5 Function `QString error()`

The plugin can provide an error message for the last failed function call. In some cases the DLT Viewer can display this message to the user. You can also just return an empty `QString` if you don't care about this.

4.6 Function `bool loadConfig(QString filename)`

The plugin can use configuration stored in files. This can be a single file or a directory containing several files. This function is called whenever a new file gets loaded.

4.7 Function `bool saveConfig(QString filename)`

Currently not used by DLT Viewer.

4.8 Function `QStringList infoConfig()`

This function is called after a call to `loadConfig()`. The plugin can provide a detailed list of the loaded configuration. This is very useful for checking whether the configuration was loaded successfully.

Chapter 5

QDLTPluginDecoderInterface description

These functions are called in the following cases:

- A new DLT message is shown in the table
- A DLT message has been filtered to create the internal filter index
- A DLT message is being exported to ASCII

5.1 Function `bool isMsg(QDltMsg &msg, int triggeredByUser)`

The plugin checks whether this DLT message should be handled by the plugin. This is called before `initMsgDecoded()` and `updateMsgDecoded()`.

5.2 Function `bool decodeMsg(QDltMsg &msg, int triggeredByUser)`

This function is only called if the previous call to `isMsg()` returned `true`. The function can access the message reference to alter its content. The function should return `true` if the message was successfully converted, and `false` otherwise.

Chapter 6

QDltPluginViewerInterface description

The following sequence of functions is called for every viewer plugin whenever a new DLT file is loaded:

- `initFileStart()`
- for each DLT message in the file:
  - `initMsg()`
  - `initMsgDecoded()`
- `initFileFinish()`

Before calling `updateMsgDecoded()`, the functions `isMsg()` and `decodeMsg()` are called for each decoder plugin, possibly altering the messages' contents.

The following sequence is called for every plugin when new DLT messages are received by DLT Viewer:

- `updateFileStart()`
- for each DLT message received:
  - `updateMsg()`
  - `updateMsgDecoded()`
- `updateFileFinish()`

Before calling `updateMsgDecoded()`, the functions `isMsg()` and is `decodeMsg()` are called for each decoder plugin, possibly altering the messages' contents.

Currently the DLT messages are iterated from the first to the last message in the DLT file. It is possible that in future versions the calls will arrive in a different order when multithreading is introduced.

6.1 Function `QWidget* initViewer()`

Called when loading the plugin. The created widget must be returned.

IMPORTANT

The widget will be owned and destroyed by the Qt framework, so don't delete it yourself.

6.2 Function `void initFileStart(QDltFile *file)`

This function is called every time a new log file is opened by the Viewer, or a new log file is created, and **before** any messages are processed with `initMsg()` and `initMsgDecoded()`.

6.3 Function void initFileFinish()

This function is called after a log file was opened by the Viewer, or a new log file was created, and **after** all messages were processed with `initMsg()` and `initMsgDecoded()`.

6.4 Function void initMsg(int index, QDltMsg &msg)

This function is called every time a new *undecoded* message is being processed when loading or creating a new log file.

6.5 Function void initMsgDecoded(int index, QDltMsg &msg)

This function is called every time a new *decoded* message is being processed when loading or creating a new log file.

6.6 Function void updateFileStart()

This function is called every time a new message was received by the Viewer and **before** the message is processed with `updateMsg()` and `updateMsgDecoded()`.

6.7 Function void updateFileFinish()

This function is called every time a new message was received by the Viewer and **after** the message was processed with `updateMsg()` and `updateMsgDecoded()`.

6.8 Function void updateMsg(int index, QDltMsg &msg)

This function is called every time a new *undecoded* message was received by the Viewer.

6.9 Function void updateMsgDecoded(int index, QDltMsg & msg)

This function is called every time a new *decoded* message was received by the Viewer.

6.10 Function void selectedIdxMsg(int index, QDltMsg &m sg)

An *undecoded* log message was selected in the message table. The Viewer plugin can now show additional information about this message.

6.11 Function void selectedIdxMsgDecoded(int index, QDl tMsg &msg)

A *decoded* log message was selected in the message table. The Viewer plugin can now show additional information about this message.

Chapter 7

QDltPluginControlInterface description

7.1 Function `bool initControl(QDltControl *control)`

This function is called once during initialisation of the plugin. It provides a pointer to the control object. Store this pointer inside your plugin for later usage.

7.2 Function `bool initConnections(QStringList list)`

This function is called when the user changes the connections to available ECUs.

7.3 Function `bool controlMsg(int index, QDltMsg &msg)`

This function is called whenever a control message is received by the DLT Viewer.

7.4 Function `bool stateChanged(int index, QDltConnection::QDltConnectionState connectionState, QString hostname)`

This function is called whenever the connection state changed for an ECU item.

7.5 Function `bool autoscrollStateChanged(bool enabled)`

This function is called whenever the *AutoScroll* option was toggled in the DLT Viewer.

Chapter 8

QDltPluginCommandInterface description

8.1 Function `bool command(QString command, QList<QString> params)`

This function gets called by the DLT Viewer framework when the **-e** command line parameter is used.