Qt for non-graphical applications

Thomas Petazzoni
Bootlin
thomas.petazzoni@bootlin.com

Thomas Petazzoni

- Embedded Linux engineer and trainer at Bootlin since 2008
  - Embedded Linux development: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux, driver development and Android system development trainings, with materials freely available under a Creative Commons license.
  - `https://bootlin.com`
- Major contributor to Buildroot, an open-source, simple and fast embedded Linux build system
- Living in Toulouse, south west of France

Agenda

- Context and problem statement
- Possible solutions
- Usage of Qt
  - Why Qt ?
  - The signal/slot mechanism
  - Usage of timers
  - Interaction with serial ports
  - Interaction with sub-processes
  - Interaction with network
  - Interaction with Linux Input devices
- Conclusion

Context and problem statement: hardware

- ARM platform
  - AT91-based
  - 400 MHz
  - 64 MB of RAM
  - 128 MB Flash
  - **No screen !**
- Platform with multiple peripherals
  - GSM modem
  - active RFID reader
  - passive RFID reader
  - GPS
  - USB barcode reader

Context and problem statement: application

Develop an application that:

- Communicates with an HTTP server over the GSM modem and/or a wired Ethernet connection
- Fetches an XML configuration from HTTP and parses it
- Handles events from RFID readers (serial port or Ethernet based) and USB barcode readers
- Manages timers per object seen through RFID (as many timers as objects seen)
- Controls the GSM modem to establish data connection and send/receive SMS
- Applies actions depending on the configuration and events
- Informs the HTTP server of events and actions happening
- Remains under a proprietary license

Possible solutions

- Write an **application in raw C**
  - Use *libcurl* for HTTP communication
  - Use *libxml2* or *expat* for XML parsing
  - Manually implement, or use another library, for basic data structure management (linked lists, hash tables, etc.)
  - As I don't like threads, use `select()` or `poll()` to handle events coming from the serial ports, the GSM modem, the network, the USB barcode reader, and the potential dozens or hundred of timers needed by the application to track objects.
- Write **a C application using *glib*** for event management and basic facilities (data structures, XML parsing, but requires *libsoup* for HTTP)
- Write an **application in Python/Ruby**
  - Quite heavy interpreter, interpreted code (no compilation), etc.
- Application had to be developed in a short period of time, and had to adapt quickly to changes in the specification of its behaviour.

Qt features

- Qt is a cross-platform toolkit for application development
- Largely used and known as a graphical widget library, but Qt is far more than that.

  - **QtCore**, event loop with an original signal/slot mechanism, data structures, threads, regular expressions
  - **QtNetwork** networking (TCP, UDP clients and servers made easy, HTTP, FTP support)
  - **QtXml** for SAX/DOM parsing of XML files
  - **QtXmlPatterns** for XPath, XQuery, XSLT and XML schema support
  - **QtGui** for GUI widgets
  - **QtMultimedia** for low-level multimedia functionality
  - **QtOpenGL**
  - **QtOpenVG**
  - **QtScript**, an ECMAScript-based scripting engine
  - **QtSQL** to query various databases
  - **QtSvg**
  - and more...

The choice of Qt

With *Qt*, we have :

- the benefits of a **compiled language**, C++ (checks at compile time, slightly better performance)
- an **ease of programming** approaching the one found in scripting languages such as Python or Ruby, with all the services provided by Qt
- a framework with an **excellent documentation**, many tutorials, forums
- a framework for which the **knowledge can be re-used** for other projects, in other situations
- a framework **licensed under the LGPL**, which allows development and distribution of proprietary applications

Moreover :

- We don't have huge performance constraints or real-time constraints
- Our platform is big enough to handle a library such as *Qt*

Size of Qt

- A common complaint about Qt could be its size
- Qt is **highly configurable**, and it is possible to build only some of the modules and for each module define which classes/features are included.
- In my application, only the following modules were needed (binary sizes given stripped for an ARM platform)
  - **QtCore**, for the event loop, timers, data structures. Weighs **2.7 MB**
  - **QtNetwork**, for HTTP communication. Weighs **710 KB**
  - **QtXml**, for parsing XML configuration files received over the network. Weighs **200 KB**.
  - No other dependencies besides the standard C++ library (**802 KB**)
- It's certainly a few megabytes, but the **ease of development** is so much higher that they are worth it, especially on a device with 128 MB of Flash and 64 MB of RAM running this single application.

Data structures: lists

Super easy chained-list mechanism, much easier to use than the `sys/queue.h` API available in C.

```
#include <QList>

QList<MyObject*> objList;

objList.append(someObj);

foreach (MyObject *obj, objList) {
   /* Do something with obj */
}

objList.removeOne(someOtherObj);

myObj = objList.takeFirst();
```

Data structures: hash maps

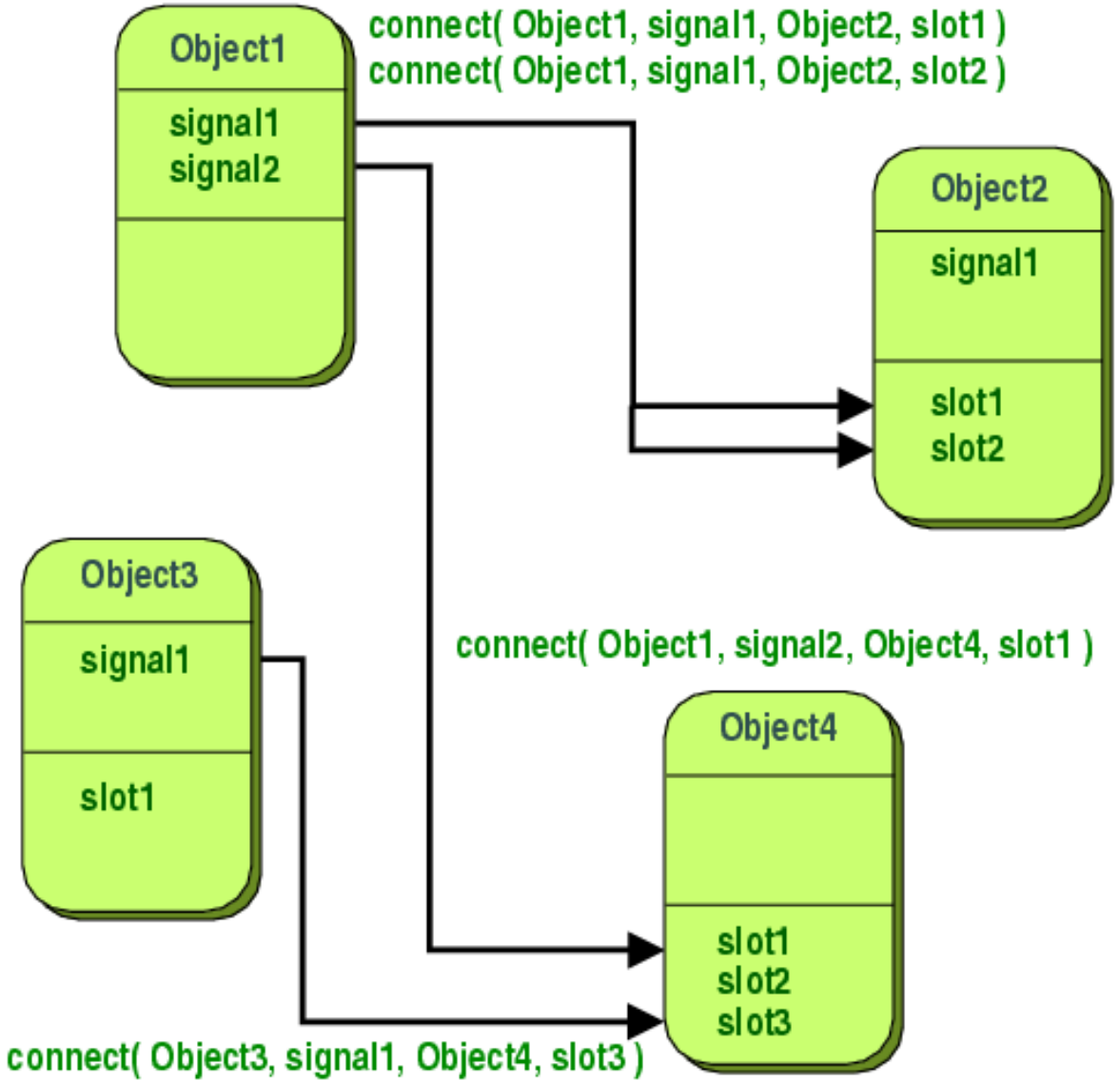
Here as well, easy to use hash maps.

```
#include <QHash>

QHash<QString, MyObject*> objMap;

QString name("someName");
objMap[name] = obj;

if (objMap.contains(name)) {
  /* Do something */
}

MyObject *obj = objMap[name];
```

Signal/slot mechanism

- The ***signal/slot*** mechanism is a core mechanism of the Qt framework
  - Objects can define ***slots***, which are functions called in response to a signal being received
  - Objects can emit ***signals***, which are events optionally associated with data
  - A signal of a given object can be **connected** to one or more prototype-compatible slots in the same object or in other objects
- Allows for very clean management of event propagation inside the application
  - Makes `select()` easy to use.
  - Most Qt classes use this mechanism, and it can be used by the application classes as well

Signal/slot diagram



Signal/slot example: slot side

A class defines a *public slot*, which is implemented as a regular C++ method.

```
class Foobar : public QObject {
 Q_OBJECT
 ...
 public slots:
   void newFrameReceived(uint identifier);
 ...
};

void Foobar::newFrameReceived(uint identifier)
{
  /* Do something */
}
```

Signal/slot example: signal side

Another class can emit a signal, and use the *emit* keyword to do so.

```
class FrameWatcher : public QObject {
 Q_OBJECT
 ...
 signals:
   void notifyFrameReceived(uint identifier);
 ...
};

void FrameWatcher::someMethod(void)
{
  uint id;
  ...
  emit notifyFrameReceived(id);
  ...
}
```

Signal/slot example: connecting things

- Connection takes place between one signal and one slot using the `QObject::connect` method.
- One signal can be connected to multiple slots.
- The great thing is that the class emitting the signal does not need to know in advance to which signal receiver classes it will be connected.

```
int main(void) {
   FrameWatcher fw;
   Foobar f;

   QObject::connect(& fw, SIGNAL(notifyFrameReceived(uint)),
                    & f,  SLOT(newFrameReceived(int)));
}
```

Timers (1/2)

- In C, timers are a bit painful to use, especially when there are dozens or hundreds of timers in various places of the application.
- In *Qt*, it's very easy and timers are naturally integrated with all other events in the event loop.

Simply define a *slot* and a *QTimer* object.

```
class Foobar : public QObject
{
 Q_OBJECT
 private slots:
   void timerExpired(void);
 private:
   QTimer timer;
}
```

Timers (2/2)

Start the timer whenever you want (here in the constructor) and your *slot method* gets called every second.

```
Foobar::Foobar()
{
 connect(& timer, SIGNAL(timeout()),
         this, SLOT(timerExpired()));
 timer.start(1000);
}

void Foobar::timerExpired(void)
{
  /* Called every second */
}
```

Spawning processes 1/2

Very easy to spawn and control sub-processes. In our application, we needed to control the execution of pppd to establish GPRS data connections.

Create a process object

```
p = new QProcess();
```

Connect its termination signal

```
connect(p, SIGNAL(finished(int, QProcess::ExitStatus)),
        this, SLOT(mySubProcessDone(int, QProcess::ExitStatus)));
```

Start the process

```
p->start("mycommand");
```

Spawning processes 2/2

Stop the process

```
p->terminate();
```

Notification of completion in a slot

```
void MyClass::mySubProcessDone(int, QProcess::ExitStatus)
{
  /* Do something */
}
```

Regular expressions

Qt has built-in support for regular expressions, there is no need for an external library such as *pcre*.

```
QRegExp r("\\[([0-9A-F]{2})([0-9A-F]{6})([0-9A-F]{2})\\]");
int pos = r.indexIn(myString);
if (pos != 0) {
     qWarning("no match");
     return;
}

uint field1 = r.cap(1).toUInt(NULL, 16);
uint field2 = r.cap(2).toUInt(NULL, 16);
uint field3 = r.cap(3).toUInt(NULL, 16);
```

Networking and HTTP

Qt also has built-in support for various network protocols, including HTTP, without the need of an external library such as *libcurl*.

Instantiate a *NetworkManager*

```
nmanager = new QNetworkAccessManager();
```

Doing a POST request

```
QNetworkRequest netReq(QUrl("http://foobar.com"));

reply = nmanager->post(netReq, contents.toAscii());
connect(reply, SIGNAL(finished(void)),
        this, SLOT(replyFinished(void)));
```

TCP server (1/3)

Defining the object implementing the TCP server

```
class MyOwnTcpServer : public QObject
{
  Q_OBJECT
public:
  MyOwnTcpServer();

private slots:
  void acceptConnection(void);
  void readClient(void);

private:
  QTcpServer *srv;
};
```

TCP server (2/3)

TCP server constructor

```
MyOwnTcpServer::MyOwnTcpServer(void)
{
  srv = new QTcpServer(this);
  srv->listen(QHostAddress::any, 4242);
  connect(srv, SIGNAL(newConnection()),
          this, SLOT(acceptConnection()));
}
```

Accepting clients

```
void MyOwnTcpServer::acceptConnection(void)
{
  QTcpSocket *sk = srv->nextPendingConnection();
  connect(sk, SIGNAL(readyRead()),
          this, SLOT(readClient()));
}
```

TCP server (3/3)

Receiving data line by line

```
void MyOwnTcpServer::readClient(void)
{
  QTcpSocket *sk = dynamic_cast<QTcpSocket *>(sender());
  if (! sk->canReadLine())
    return;
  char buf[1024];
  sk->readLine(buf, sizeof(buf));
  /* Do some parsing with buf, emit a signal
     to another object, etc. */
}
```

XML Parsing

Instantiating a DOM document with file contents

```
QFile *f = new QFile();
f->open(QIODevice::ReadOnly | QIODevice::Text);

dom = new QDomDocument();
dom->setContent(f);
```

- And then, thanks to the `QDomDocument`, `QDomNodeList`, `QDomNode` and `QDomElement` classes, you can easily parse your XML data.
- Not much different from *libxml2*, but it's built into *Qt*, no need for an external library.
- Of course, besides the basic *DOM* API, there is also a *SAX* API and a special *stream* API.

Communicating with serial ports (1/4)

- Communicating with serial ports was essential in our application
- Of course, using the classical C API is possible, but we would like to integrate serial port communication into the *Qt* event loop
- The *QExtSerialPort* additional library makes this really easy.
- `http://code.google.com/p/qextserialport/`

Communicating with serial ports (2/4)

Initialization

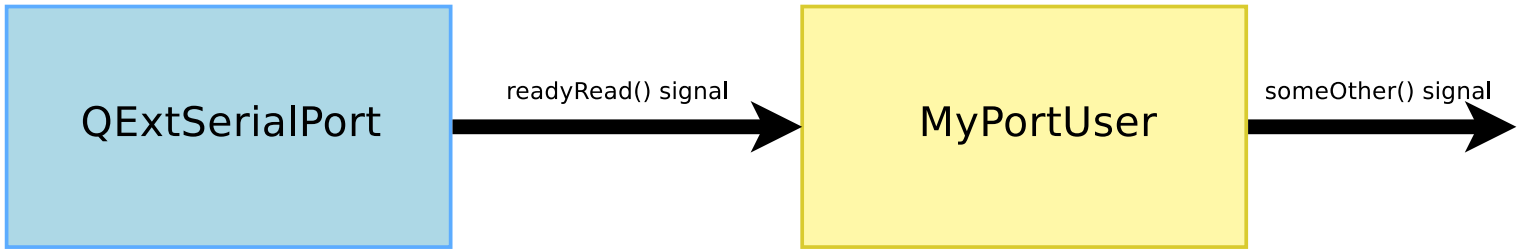
```
port = new QextSerialPort("/dev/ttyS3",
                          QextSerialPort::EventDriven);
port->setBaudRate(BAUD9600);
port->setFlowControl(FLOW_OFF);
port->setParity(PAR_NONE);
port->setDataBits(DATA_8);
port->setTimeout(0);
port->open(QIODevice::ReadOnly);
connect(port, SIGNAL(readyRead()),
        this, SLOT(getData()));
```

Communicating with serial ports (3/4)

Receiving data in the slot method

```
void MyClass::getData(void)
{
  while (1) {
    char c;
    if (port->read(& c, 1) <= 0)
      break;

    /* Do something, like parse the received data, and
       emit a signal when something meaningful has been
       received */
  }
}
```

Communicating with serial ports (4/4)



Using Linux input devices (1/4)

- In the project, we had to use USB barcode readers, which are implemented as standard USB HID devices
- *QtGui* obviously has support for a wide range of input devices
- But in *QtCore*, there is no dedicated infrastructure
- So we wanted to integrate the event notification of a Linux Input device into the *Qt* event loop
- This is very easy to do with `QSocketNotifier`

Using Linux input devices (2/4)

Class declaration

```
class QLinuxInputDevice : public QObject
{
Q_OBJECT
public:
  QLinuxInputDevice(const QString &name);

signals:
  void onInputEvent(struct input_event ev);

private slots:
  void readyRead(void);

private:
  int fd;
  QSocketNotifier *readNotifier;
};
```

Using Linux input devices (3/4)

Initialization

```
QLinuxInputDevice::QLinuxInputDevice(const QString &name)
{
  fd = ::open(fileName.toAscii(), O_RDWR | O_NONBLOCK);

  readNotifier =
    new QSocketNotifier(fd, QSocketNotifier::Read, this);

  connect(readNotifier, SIGNAL(activated(int)),
          this, SLOT(readyRead()));
}
```

The `QSocketNotifier` mechanism tells *Qt* to add our file descriptor in its `select()` loop and to dispatch events on this file descriptor as *Qt* signals.

Using Linux input devices (4/4)

Receiving events

```
void QLinuxInputDevice::readyRead(void)
{
  struct input_event ev[64];

  while(1) {
    int sz = ::read(fd, ev, sizeof(ev));
    if (sz <= 0)
      break;
    for (int i = 0;
         i < (sz / sizeof(struct input_event));
         i++)
      emit onInputEvent(ev[i]);
  }
}
```

Integrating Unix signals 1/2

We want to get a nice *Qt* signal when an *Unix* signal is received. The goal is to handle *SIGHUP* in order to reopen the log file (after it has been rotated by *logrotate*).

Class definition

```
class QUnixSignalHandler : public QObject
{
  Q_OBJECT
public:
  QUnixSignalHandler(int signal);

signals:
  void fired(void);

private slots:
  void gotSignal(void);

private:
  int fd;
};
```

Integrating Unix signals 2/2

To implement this, we use the *signalfd()* system call and a *QSocketNotifier*. Thanks to this, the Unix signal notification is completely integrated into *Qt* event loop, like all other events.

Class implementation

```
QUnixSignalHandler::QUnixSignalHandler(int signal)
{
  sigset_t sigset;
  sigemptyset(& sigset);
  sigaddset(& sigset, signal);

  sigprocmask(SIG_BLOCK, &sigset, NULL);

  fd = signalfd(-1, & sigset, 0);
  if (fd < 0)
    qFatal("Bork");

  QSocketNotifier *sn = new QSocketNotifier(fd, QSocketNotifier::Read, this);
  connect(sn, SIGNAL(activated(int)), this, SLOT(gotSignal()));
}

void QUnixSignalHandler::gotSignal(void)
{
  struct signalfd_siginfo si;
  ::read(fd, & si, sizeof(si));
  emit fired();
}
```

Usage of Unix signal integration

Our *Logger* class instantiates a *QUnixSignalHandler* and connects its *fired()* signal to a slot that reopens the log file.

Example of *QUnixSignalHandler* usage

```
Logger::Logger(QString _logFile)
{
        [...]
        sigHandler = new QUnixSignalHandler(SIGHUP);
        connect(sigHandler, SIGNAL(fired()),
                this, SLOT(reOpenLogFile()));
}

void Logger::reOpenLogFile(void)
{
        [...]
}
```

Coming back to the signal/slot mechanism

- The *signal/slot* mechanism is really a **great** feature of *Qt*
- It unifies all the event management into something much easier to use than a single, central, `select()` event loop
- All events, such as timer expiration, communication on serial ports, on TCP sockets, with Linux input devices, communication with external process are all handled in the same way.
- Each class appears to do its own *event management*, locally, making the code very straight-forward
- Allows to easily write a single-threaded application: avoids the need for threads and complicated mutual exclusion, synchronization issues, etc.
- A bit more difficult to manage the the GPRS modem, needed a moderately elaborate state machine.

Building Qt applications

- *Qt* comes with its own build system, called **qmake**.
  - basic and easy to use build system
  - a `.pro` file describes the project sources and required libraries, and a Makefile is automatically generated by *qmake*
  - a specially-configured *qmake* is needed to do cross-compilation.
  - `http://doc.qt.nokia.com/latest/qmake-manual.html`
- For a more powerful build system, **CMake** is definitely a good choice
  - the one we have chosen for our project
  - the developer writes a `CMakeLists.txt` file, `cmake` parses this file, runs the associated checks and generates a conventional *Makefile*
  - `http://www.cmake.org`.

Building with *Cmake* : `CMakeLists.txt`

```
project(myproject)

find_package(Qt4 REQUIRED COMPONENTS QtCore QtNetwork QtXML)

set(myproject_SOURCES main.cpp file1.cpp file2.cpp file3.cpp)
set(myproject_HEADERS_MOC file1.h file2.h)
set(myproject_HEADERS file3.h)

QT4_WRAP_CPP(myproject_HEADERS_MOC_GENERATED
             ${myproject_HEADERS_MOC})

include(${QT_USE_FILE})
add_definitions(${QT_DEFINITIONS})
```

Note: newer versions of *CMake* will make it even easier, the `QT4_WRAP_CPP` call and related things due to the *moc* pre-processor are done implicitly.

Building with *Cmake* : CMakeLists.txt

```
find_package(PkgConfig)
pkg_search_module(EXTSERIALPORT REQUIRED qextserialport)
link_directories(${EXTSERIALPORT_LIBRARY_DIRS})
include_directories(${EXTSERIALPORT_INCLUDE_DIRS})

add_executable(myapp ${myproject_SOURCES}
                ${myproject_HEADERS_MOC_GENERATED}
                ${myproject_HEADERS})
target_link_libraries(myapp ${QT_LIBRARIES}
                      ${EXTSERIALPORT_LIBRARIES})

install(TARGETS myapp
  RUNTIME DESTINATION bin
)
```

See also http://developer.qt.nokia.com/quarterly/view/using_cmake_to_build_qt_projects.

Building with *CMake*

- Create a build directory

  `mkdir build; cd build`

- Start the configuration process

  ```
  cmake
  -DCMAKE_TOOLCHAIN_FILE=/path/to/toolchain.file
  /path/to/sources
  ```

  - The *toolchain file* describes your toolchain, and can be generated by your embedded Linux build system. Makes the cross-compilation process very easy.

- Compilation

  `make`

- Installation

  `make DESTDIR=/path/to/rootfs install`

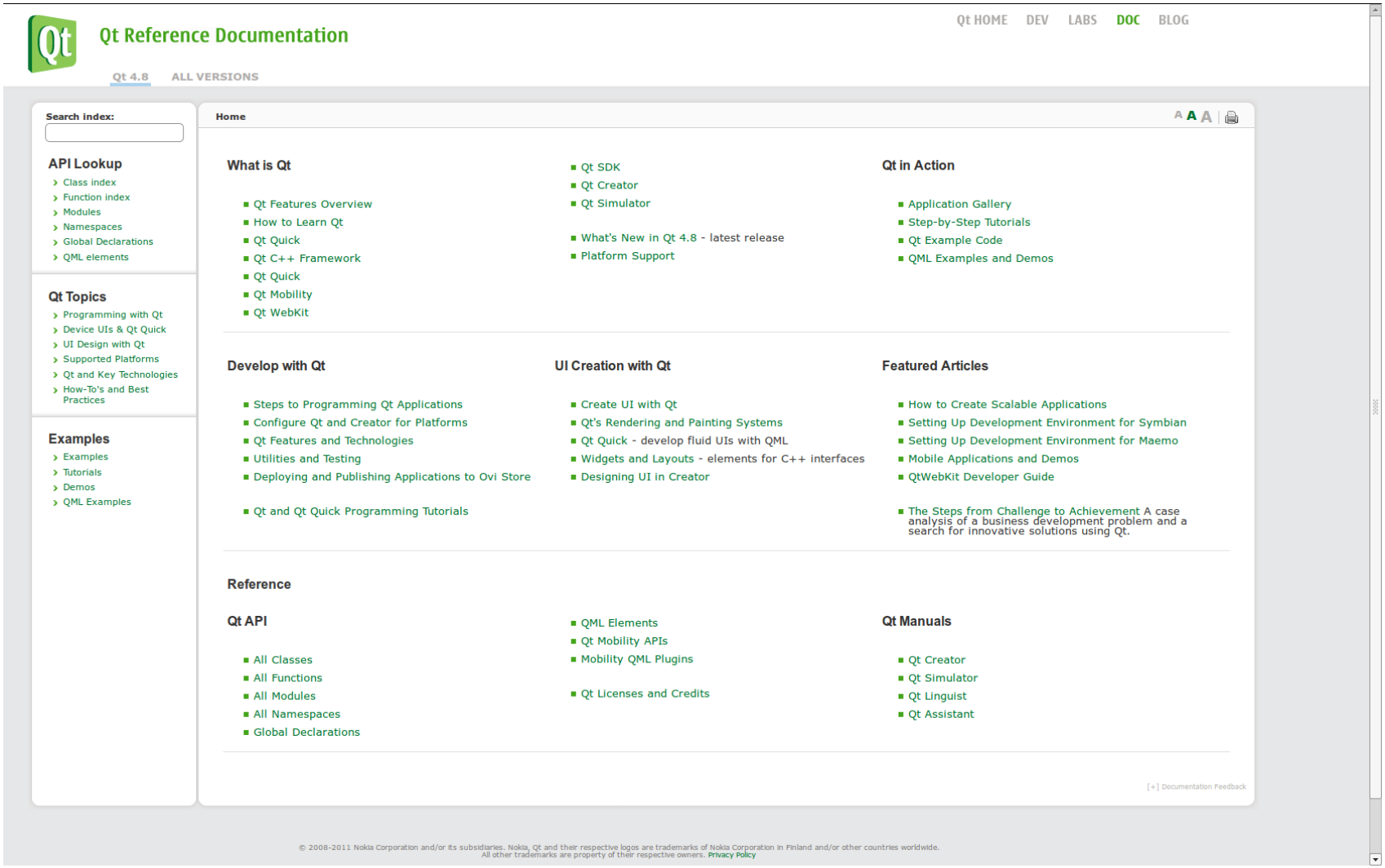
Qt and embedded Linux build systems

Qt is packaged is many embedded Linux build systems, which makes it quite easy to use and integrate in your system:

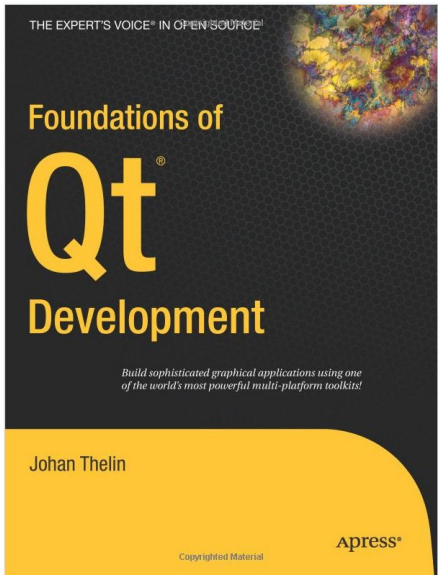
- **Buildroot**, my favorite. Possible to select at configuration time which *Qt* components should be built. Also generates a *CMake* toolchain file.
- **OpenEmbedded/Yocto**. Always builds a full-blown Qt, but possible to install only some parts to the target root filesystem.
- **OpenBricks**, same thing.
- **PTXDist**, also allows to customize the *Qt* configuration to select only the required modules.

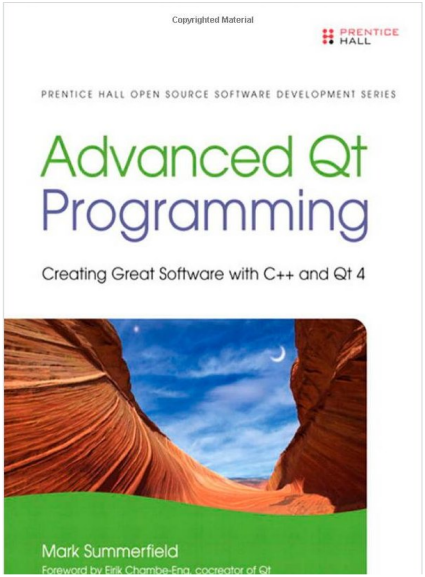
Documentation

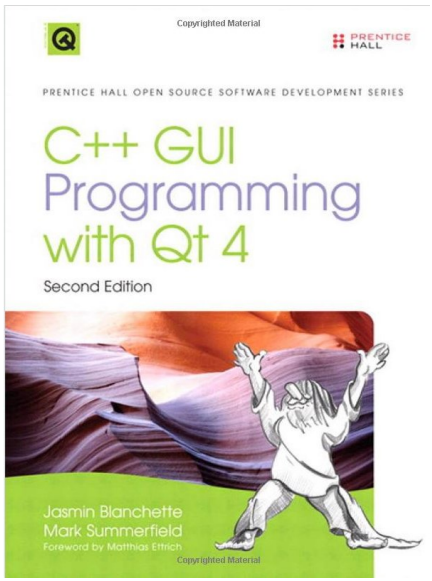
Excellent documentation: both reference documentation and tutorials, at `http://doc.qt.nokia.com/`.

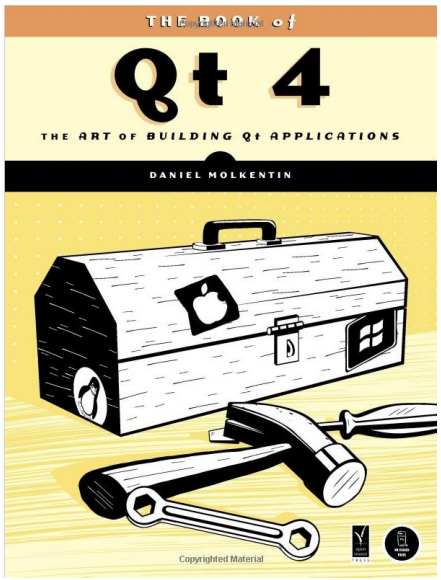


Books


THE EXPERT'S VOICE® IN OPEN SOURCE
Foundations of
Qt®
Development
Build sophisticated graphical applications using one of the world's most powerful multi-platform toolkits!
Johan Thelin
Apress®
Copyrighted Material


Copyrighted Material
PRENTICE HALL
PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES
Advanced Qt Programming
Creating Great Software with C++ and Qt 4
Mark Summerfield


Copyrighted Material
PRENTICE HALL
PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES
C++ GUI Programming with Qt 4
Second Edition
Jasmin Blanchette
Mark Summerfield
Foreword by Matthias Ettrich
Copyrighted Material


THE BOOK of
Qt 4
THE ART OF BUILDING Qt APPLICATIONS
DANIEL MOLKENTIN
Copyrighted Material

Issues encountered

The development of our application went very smoothly, and *Qt* allowed to focus on the application itself rather than little "details". Our only issues were:

- The fully *asynchronous* paradigm of signal/slot makes it a little bit involved to write purely sequential segments of code. Our case was managing the GPRS modem, which involves sending multiple `AT` commands in sequence. We had to write a moderately elaborate state machine for something that is in the end quite simple.
- The single-threaded architecture is very sensitive to delays/timeouts. By default, the *QExtSerialPort* configures the serial ports to wait 500ms before giving up reading (*VTIME* termios). This caused our application to loose events from *LinuxInput* devices.

Conclusion

Qt really met all its promises:

- It was easy to learn, even for a very moderately-skilled C++ programmer (me !)
- It allowed to develop the application very quickly by focusing on the application logic rather than "low-level" details (XML parsing, HTTP requests, event management, etc.)
- Ease of programming very close to Python. You code something, and it just works.
- The skills have already been re-used for the development of another Qt application, this time a graphical demo application for a customer.

I would therefore strongly recommend considering the use of *Qt* amongst the possible frameworks for your applications, even non-graphical ones.

Questions?

Thomas Petazzoni

thomas.petazzoni@bootlin.com