Qualcomm Gobi devices in Linux based systems

Aleksander Morgado
aleksander@lanedo.com

Lanedo

December 10, 2013

1 Introduction

For a very long time, mobile broadband modems have been controlled using mobile-specific AT protocol extensions. These extensions were usually defined by the implementors of the mobile broadband technology standards (e.g. the European Telecommunication Standards Institute, ETSI GSM 07.07, later 3GPP TS 27.007[1]), or by device vendors themselves (e.g. Qualcomm extensions for CDMA/EV-DO devices). These modems would inherit most of the logic that was applicable to dial-up modems. This includes the AT protocol itself, but also the requirement of a PPP[2] session to be established to serve as an interface between the computer and the modem (or even the network itself).

Once requirements in the connection throughput were increased (e.g. LTE/4G data rates), and newer technologies appeared to interface between a device and the host computer (e.g. USB), manufacturers started to implement their own new protocols to control mobile broadband devices. The AT+PPP pair became just a legacy approach which is still available in most of the devices, but purely for compatibility with systems which do not talk the new protocols yet.

One of those new protocols implemented to control mobile broadband modems is the *Qualcomm MSM Interface* (QMI) protocol developed by Qualcomm, which can be used in Linux kernel based operating systems through two mutually exclusive kernel drivers: '*GobiNet*' and '*qmi_wwan*'. These two drivers take completely different approaches to handle the protocol; GobiNet is a very complex driver which implements within the kernel most of the core protocol logic, while qmi_wwan just leaves all those tasks to user-space processes (therefore keeping the kernel bits as small as possible).

The purpose of this document is to give a primer of the QMI protocol, and show the differences, advantages and drawbacks of using either the GobiNet or qmi_wwan kernel drivers and how user-space applications interact with them.

Kernel driver	Owner	Project	User-space
GobiNet	Qualcomm	Code Aurora	GobiAPI
qmi_wwan	Community	Linux kernel	libqmi, ofono, uqmi, ...

[1] http://www.3gpp.org/DynaReport/27007.htm
[2] Point-to-Point Protocol

2 Gobi and QMI

2.1 Gobi modules

Gobi[3] modules are mobile broadband modems designed by Qualcomm, although devices with these chipsets are also often manufactured by other vendors, like Sierra Wireless or Novatel. These modems are able to work with very different 2G/3G/4G radio access technologies, including those defined by both 3GPP (GSM, GPRS, UMTS, HSPA, LTE, ...) and 3GPP2 (CDMA-1x, EV-DO, ...).

Unlike many other mobile broadband modems, and even if the devices actually expose AT-capable serial ports, Gobi modules are not designed to be controlled with AT commands. Instead, Qualcomm designed a new *Qualcomm MSM Interface* (QMI) binary protocol to communicate with the modem. This protocol can make use of the USB subsystem (among others) to transfer requests, responses and indications between the host and the device, and therefore doesn't rely on a serial port, as is the case with AT and DM/DIAG[4].

The other major advantage of relying on the USB subsystem is the fact that there is no longer the need to setup a PPP session over a serial port to get a connection established. Instead, packets can be sent and received over an ECM/NCM[5] like USB interface exposed by the device. The benefits of using a real network interface instead of PPP over a serial port are not only about being easier to manage; the actual throughput achievable with a network interface is much higher as the overhead introduced by PPP is fully avoided. In fact, all modems that need to handle the data rates available with a technology like LTE do require such a network interface.

2.2 The QMI protocol

The QMI protocol defines different **services**, each of them related to different actions that may be requested from the modem. For example, the *DMS* (Device Management) service provides actions to load device information, the *NAS* (Network Access) service provides actions to register in the network, and the *WDS* (Wireless Data) service allows setting up IP connections. Before using a service, the user needs to handle the creation of **clients** for those services by allocating/deallocating *client IDs* using the generic always-on *CTL* (control) service.

Each service in the protocol defines **Requests and Responses** as well as **Indications**. Each pair of request and response has a matching ID which lets the user concatenate multiple requests and get out-of-order responses that can be matched through the common ID afterwards. Indications arrive as unsolicited messages, sent either to a specific client or as a broadcast message to all the clients of a given service.

Finally, each message in the protocol defines a set of input (in requests) and output (in responses and indications) arguments, specified as **Type-Length-Value** (TLV) fields. Some of these are defined to be mandatory, others are optional, and others are only mandatory if some other argument is available and has a given value. The Type of the field defines not only which field it is, but also the format of the value to be expected (e.g. an integer, or a string, or some other supported format).

Just this level of organization outlined in the previous paragraphs makes the communication with the modem much more clear, as the messages are transferred in an organized way and are self-contained packs of information with a predefined format. So, even if more complex, it is far more powerful and less error-prone than sending and receiving text-based AT strings via a serial port.

[3] http://www.qualcomm.com/gobi
[4] DM/DIAG is an older binary protocol also developed by Qualcomm
[5] Ethernet Control Model/Network Control Model

3 Kernel space

Being able to control Gobi modules in Linux-based operating systems has been possible for a long time now, using the '**GobiNet**' GPL/BSD dual-licensed kernel driver developed by Qualcomm, which is published under the Linux Foundation's CodeAurora[6] initiative. In late 2010, there was an attempt[7] to have in the upstream kernel a Linux-styled rewrite of the original driver, but it never got through for different reasons.

The biggest concern of the Linux kernel developers was that the GobiNet driver does much more than what a usual Linux kernel driver does. A software driver should, if possible, just provide access to the capabilities of a hardware device in a generic manner; or in other words, using the device in one way or another is a task of user-space processes, not a task of the kernel. Also, the smaller the driver is the more robust the system is, as code running in kernel space can cause errors in the whole system.

In order to handle the issue, upstream kernel developers took a fully different approach and developed instead a new '**qmi_wwan**' driver, simpler in orders of magnitude and much easier to maintain, while leaving all the complex logic to user-space processes, which are usually more flexible and where it is easier to change behavior or fix errors. This new driver has been available since Linux 3.4.

This conflict between kernel drivers is uncommon because of how the QMI protocol is defined. It's harder to draw the line between what should be kernel or user space based tasks. The following sections try to detail where that line was drawn in both approaches.

3.1 Gobi as a USB device

A Gobi module connected to a Linux-based system via a USB connection will be exposed as any other standard USB device, with multiple interfaces and multiple endpoints for each interface[8]. Each of the interfaces exposed serves a different purpose: one of the interfaces gives may access to a serial port, while another one gives access to the QMI port and its corresponding net port. It is worth noting that different devices will expose the QMI-capable interface with different interface numbers, so the kernel will always need to have a table of rules which matches a given device (e.g. given its USB vendor ID or product ID) with its corresponding QMI-capable interface number[9].

A USB interface which is capable of providing both QMI and network packet transfers will expose four endpoints:

- CONTROL: The default bi-directional endpoint, which will be used to send and receive QMI messages embedded within *SendEncapsulatedCommand* and *GetEncapsulatedResponse* requests as defined by the CDC[10] USB class.
- INTERRUPT IN: The endpoint used by the device to notify that incoming QMI messages can be read.
- BULK IN: Bulk endpoint for IP packets sent from the device to the host.
- BULK OUT: Bulk endpoint for IP packets sent from the host to the device.

It is worth noting that this previous interface layout is the most common one, but not unique. Some devices may export the QMI and net capabilities using different interfaces, specifying which are the expected ones in a *Union CDC functional descriptor* on the control interface.

[6] https://www.codeaurora.org
[7] http://thread.gmane.org/gmane.linux.network/174190
[8] For completeness, interfaces are also grouped in *configurations*, and each interface may have different *alternate* settings.
[9] If multiple *alternate* settings are given for an interface, the kernel will also need to decide which one provides access to the required endpoints.
[10] Universal Serial Bus Class Definitions for Communication Devices

3.2 Non-USB QMI devices

Qualcomm Snapdragon[11] processors include both the CPU and the mobile broadband telephony stack on the same chip (System-On-Chip[12]); i.e. there isn't a separate chip implementing the 3G/4G connectivity. In this kind of setup, the communication between the kernel and the stack is done using shared memory (SMD), instead of relying on a USB connection. This setup is left out of the scope of this document, as there is little information about how this is really managed.

3.3 GobiNet

As previously noted, GobiNet is the driver developed by Qualcomm to manage QMI-capable modems, and is not available in the upstream Linux kernel. Being an out-of-tree module wouldn't be so problematic if the code base was updated for each new kernel release (important if USB driver changes are made upstream) and accessible all the time. Sadly, the source code repository is private and the driver releases are not aligned to the upstream kernel ones. This issue alone comes with several side-effects:

- Users wanting to install the GobiNet driver will, very likely, need to adapt the code of the driver for the kernel they are using; unless they use a kernel which was released before the actual driver and the developers included support for that kernel (e.g. back-porting features and bug fixes from future kernels releases they may want or need). It is obvious that not every user is a kernel developer, not even a C developer, so the changes being done in the user's tree have no peer review or testing for correctness or stability before used.
- The published driver supports not only the latest kernel at that time; it also should support kernels released before. When a driver is maintained within the upstream kernel tree this is of course not needed. Needing to support previous kernel versions make the code of the driver full of code sections which are only applicable to different kernel versions, increasing its complexity.
- The GobiNet driver is maintained by Qualcomm, and only Qualcomm manufactured devices are handled, and not even all available ones. Adding support for a new device requires, at least, to modify the USB vendor ID and product ID table of allowed devices in the GobiNet driver itself.
- Every other non-Qualcomm manufacturer which produces modems based on Gobi chipsets ends up providing to their users a different version of the GobiNet driver; e.g. adjusting the expected layout of the USB interfaces, and obviously including their own set of VID/PIDs.

3.3.1 Architecture

The GobiNet driver takes care of all the four endpoints exposed in the QMI/net interface, and sets up two devices to be available in user-space (see Figure 1).

The driver creates a **/dev/qcqmiX** character device which acts as the control interface of the modem, handling the QMI protocol. It will take care of receiving QMI messages from user-space, packing them into *SendEncapsulatedCommand* requests and sending them to the modem through USB. In the same way, it will receive QMI messages from the modem with *GetEncapsulatedResponse* requests, and it will send them back to user-space. The driver will use both the CONTROL and INTERRUPT endpoints for this purpose.

The driver also collects both BULK IN and BULK OUT endpoints and uses the *usbnet* generic driver in the kernel to expose a **usbX** network interface. The GobiNet driver can be seen as inheriting from usbnet, i.e. providing the same functionality and reusing parts of usbnet, but handling the internal logic in a way specific to Gobi modules.

[11] http://www.qualcomm.com/snapdragon
[12] http://en.wikipedia.org/wiki/System_on_a_chip

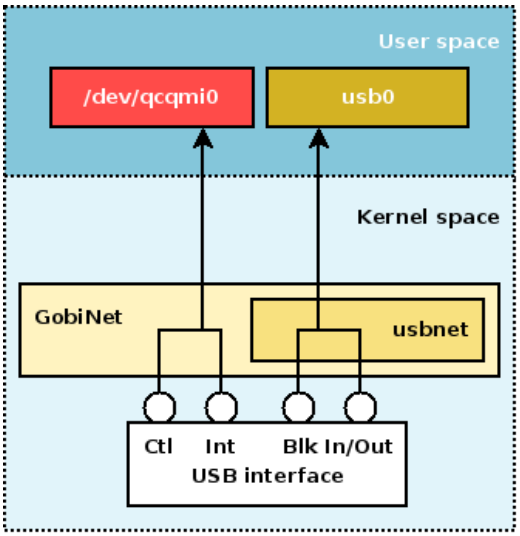

Figure 1: GobiNet in the Linux kernel

3.3.2 The CTL service

The most important *feature* of the GobiNet kernel driver is that it completely hides the usage of the *CTL* (control) QMI service to user-space. Every action that can be performed by the CTL service will be done exclusively by the driver, including:

- **Version info:** Querying which QMI service versions are supported by the device.
- **Client allocation:** Allocating different *Client IDs* for different services.
- **Client release:** Releasing the *Client IDs* that were previously allocated.
- **Data format:** Configuring the format of network packets to be transferred through the data network interface.

The whole driver is organized around the need to hide this service to user-space, and therefore it requires a high complexity to handle all the different features provided by the protocol. For example, the driver needs to take care of forwarding *Responses* and non-broadcast *Indications* only to the user-space process which acquired a given *Client ID* for a given service. But, from a different perspective, this also means that multiple user-space processes can share the access to a single QMI device without interferring with each other.

3.3.3 Device initialization

Upon detection of a new QMI capable modem, the kernel driver will perform several actions before exposing the character and network devices to user-space. One of such operations is making sure that the modem is ready to talk QMI, which is accomplished by a periodic transmission of a *CTL Version Info* request until the modem replies with a correct response. Once the modem replies correctly, the driver will also create a temporary *DMS* (Device Management Service) client to gather the MEID[13] of the hardware modem (which will make it available through an ioctl command), and will also set up an internal *WDS* (Wireless Data Service) client to gather network transfer statistics (which will be made available directly in the network interface).

3.3.4 Client allocation and releasing

As previously explained, a user wanting to use the QMI protocol needs to allocate a *Client ID* for each QMI service to be used. In the case of the GobiNet driver, QMI client allocation and releasing is done by the kernel driver itself, and follows this sequence of events:

[13] Mobile Equipment Identifier

1. The user opens the /dev/qcqmiX character device.
2. The user runs a driver-defined ioctl command on the open device to request a new *Client ID* for a given service.
3. The kernel driver sends a *CTL Client Allocation* request to the modem, which replies with a newly allocated unique ID for the service requested.
4. The kernel returns the *Client ID* as the result code of the ioctl command, although the user can choose not to store it. From the user point of view, the returned value is just an indication of whether the allocation succeeded or failed. Once a *Client ID* has been allocated for an open /dev/qcqmiX character device, only the specific service that was requested can be used in this instance.
5. The user can now send QMI *Requests* for the requested service, and receive QMI *Responses* and *Indications*. Given that the *Client ID* and service values are also maintained in the driver, and that the open instance is exclusive for the *Client ID* and service pair, the user doesn't really need to fill in those values when creating a QMI *Request*, the driver will take care of that before sending the actual fully valid message to the modem.
6. When no longer needed, the user closes the character device.
7. The kernel will then perform a *CTL Client Release* itself, releasing the previously allocated *Client ID*.

This logic to manage multiple concurrent clients forces a process which requires multiple services to open the /dev/qcqmiX character device multiple times, one for each needed service. But of course, it also allows multiple processes to do the same operations at the same time.

Another side effect of managing the *Client ID* within the kernel driver is that if a process ends unexpectedly, the kernel can still release the allocated *Client ID*s, skipping the problem of leaving 'leaked' unreleased ones. All Gobi devices have a limit of client IDs that can be allocated simultaneously, so running out of these is a serious issue[14].

3.4 qmi_wwan

Since Linux 3.4, the qmi_wwan driver has been available in the upstream Linux releases. Being a driver maintained within the upstream tree means that none of the problems exposed for GobiNet are applicable, so:

- Users will get the driver already available for the exact kernel they are using.
- Most distributions already enable the qmi_wwan driver in their default builds.
- The Linux tree contains always the latest sources for the latest kernel. There are no conditional code segments with path ways for separate versions of older kernels - which tends to add complexity and a maintenance burden. This keeps the code leaner and cleaner generally speaking.
- The qmi_wwan driver supports multiple QMI capable devices, regardless of which manufacturer produced them. The driver takes care of detecting the USB layout and gathering the correct endpoints from the correct interfaces. Therefore, there is no need for vendor-maintained versions of the driver.

Needless to say that the qmi_wwan driver gets updated with new supported devices soon after being released. Usually users with new modems will report the new VID/PIDs to use, or even manufacturers will notify the Linux kernel community about the new devices before even being sold to the public.

3.4.1 Architecture

The qmi_wwan is very similar to the GobiNet driver from an architectural point of view (see Figure 2), which is obvious as the task of both drivers is to provide both a control and data interface in userspace.

[14] Gobi3000 devices, though, have the ability to reset and release all Client IDs via a *CTL Sync* command.

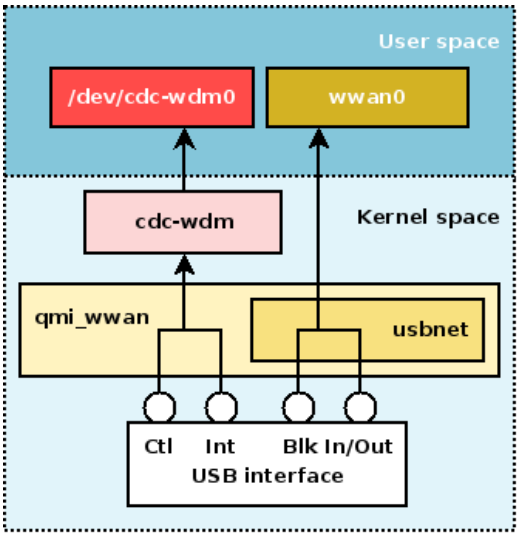

Figure 2: qmi_wwan in the Linux kernel

QMI over USB basically uses CDC-WDM messages for the transport, and since the kernel already has code for that, the qmi_wwan driver fully relies on the generic *cdc-wdm* driver to manage the CONTROL and INTERRUPT endpoints.

This is possible because, as explained earlier, the USB communication is done using *SendEncapsulatedCommand* and *GetEncapsulatedResponse* requests as defined by the CDC USB class. The cdc-wdm driver supports the WMC Device Management functionality of cell phones compliant to the CDC WMC specification[15]. The result of using the cdc-wdm as a subdriver in qmi_wwan is that a new **/dev/cdc-wdmX** character device is exposed for user-space use, which allows exchanging QMI messages.

As with the GobiNet case, qmi_wwan also exposes a network interface, named as **wwanX** because the driver gets explicitly registered with the FLAG_WWAN flag.

3.4.2 The CTL service

The qmi_wwan driver knows nothing about the existence of a *CTL* service. It doesn't even know that QMI is the protocol being used in the messages transferred between user-space and the modem. But leaving all the hard work to user-space doesn't come without any cost. In particular, relaying to user-space the management of the actions performed by the *CTL* service means that only one single process will be able to successfully use the **/dev/cdc-wdmX** character device[16].

3.4.3 Device initialization

The qmi_wwan driver will not do any device initialization before exposing the new character and network devices in user-space. In particular, a user-space process wanting to use the QMI port will need to explicitly perform checks to see whether the device is ready to be used. Also, no connection statistics will be automatically gathered from the device.

3.4.4 Client allocation and releasing

Following the simplicity required by the driver, and given that the driver knows nothing about the QMI protocol itself, there is no built-in mechanism to handle QMI service client allocations. Every user-space process

[15] USB CDC subclass for Wireless Mobile Communication devices

[16] To handle this issue, the libqmi implementation in user-space introduces a 'qmi-proxy' daemon, which is the only one accessing the QMI port.

wanting to use QMI, will need to handle the allocation and releasing of *Client IDs* using the *CTL* service.

Leaving the *Client ID* releasing task to user-space also makes it impossible to make sure that all *Client IDs* are released when a process ends. A well implemented program using the QMI protocol should try to perform the *Client ID* releases before exiting, but it is assumed that unexpected terminations of the running process will end up leaving 'leaked' *Client IDs* which are never released.

This limitation, though, provides itself a very useful feature which was not possible before with the GobiNet driver. The fact that a program can choose whether or not to release a *Client ID* when exiting means that it can also choose to reuse a *Client ID* that was maybe allocated in a previous run of the same program. From a practical point of view, the fact that the kernel doesn't automatically release the clients when the process exits allows users to use the QMI protocol from within scripts or the command line, as there is no need to allocate or release a client in each run of the script or command line program[17]. For most operations that can be performed through QMI, allocating/releasing a new client for each operation is a possibility, but there are some very specific use cases (e.g. *WDS Start Network* to setup a network connection) which require the *Client ID* to be kept registered all the time (or the connection will be dropped).

It is worth noting again that avoiding complex operations like these in the kernel driver is by no means a *limitation* of the driver; instead, the driver is kept as simple as possible and therefore much more robust. A simple driver is also much more flexible, as user-space processes can process *all* the logic of the protocol, not just parts of it.

[17] Which is exactly what libqmi's *qmicli* and *qmi-network* tools allow to do.

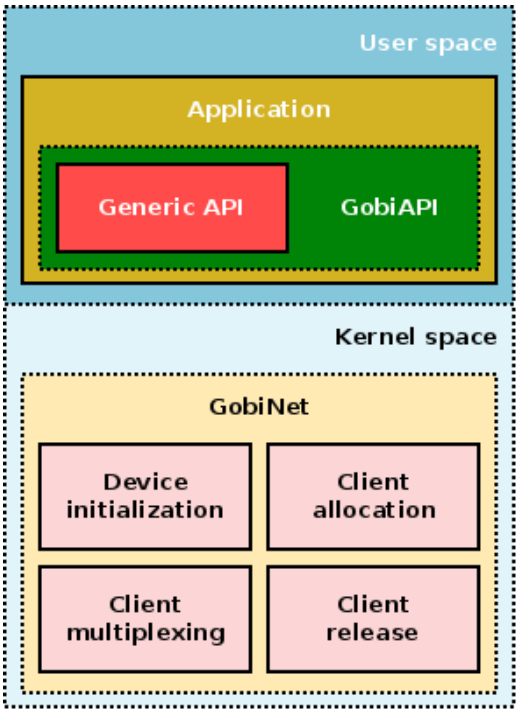

Figure 3: GobiAPI in user-space

4 User space

It should be obvious by now that while using the control character device exposed by the GobiNet driver is quite straightforward for user-space processes, using the one exposed by qmi_wwan is a much more complex task. Still, there are already free and open source projects out there that make it equally easy - if not easier - to use the qmi_wwan driver.

This section shows some of the currently available software implementations using the GobiNet and qmi_wwan kernel drivers.

4.1 GobiAPI using GobiNet

Along with the GobiNet driver, Qualcomm also implemented and published the **GobiAPI** library (see Figure 3). This library, developed in C++, provides a high level access to the features of a Gobi module, completely hiding to the user of the library how the communication with the actual USB device is done. This of course includes the ioctls required to allocate *Client IDs*, as well as the logic to asynchronously receive QMI messages from the device (using multiple threads).

In addition to the generic code to communicate with the control character device, this library also exports methods to both parse received *Responses* and *Indications* as well as to create new *Requests*. These methods are also be very heavily based on a set of structures that map to the possible *Type-Length-Value* fields.

This library was designed to work exclusively with the GobiNet driver, and is also maintained in the CodeAurora[18] project.

4.1.1 Device initialization

The GobiAPI fully relies on the GobiNet kernel driver to make sure that a QMI device is fully initialized before being used. The kernel will only expose the /dev/qcqmiX character device for fully initialized devices.

[18] https://www.codeaurora.org

4.1.2 Client allocation and releasing

As explained in the previous sections, the user-space process doesn't need to manage the Client allocation and releasing logic itself. The GobiAPI will just perform an ioctl to request a *Client ID* to be allocated when it opens the port, and will not even store this value anywhere (as GobiNet in the kernel does it). A program using the GobiAPI will just need to make sure that for each QMI service it wants to use it opens a new file descriptor from the QMI control port.

4.1.3 Client multiplexing

Given that multiple concurrent access to the port is already provided by the GobiNet driver in the kernel, the GobiAPI library doesn't need to do anything special to make sure that the QMI commands sent by one process don't collide with the ones from another process.

4.2 libqmi using qmi_wwan

libqmi[19] is a free and open source library developed by free software contributors, and managed in the freedesktop.org[20] public repositories. Unlike the GobiAPI implementation, libqmi exposes a code source repository with the latest fixes available, which makes it perfect to be able to get the most recent improvements and fixes.

Most Linux-based distributions already include packages for libqmi, and it is also used by **ModemManager**[21], a mobile connection manager widely used along with NetworkManager[22].

This C library relies on the GLib[23], GObject[24] and GIO[25] libraries to provide a very high level interface to interact with QMI modems when the qmi_wwan kernel driver is being used. The generic API[26] exported by the library is a bit different from the one exported by GobiAPI, as this library not only handles the actual creation and parsing of QMI messages, but also the full *CTL* service, including the device initialization sequence and the whole *Client ID* allocation and releasing logic (see Figure 4).

4.2.1 Device initialization

A program wanting to use the /dev/cdc-wdmX character device for QMI control will be able to specify that a 'version check' is to be run during the opening sequence of the device. This check will issue a periodic transmission of a *CTL Version Info* request until the modem replies with a correct response, so it is equivalent to the one done by the GobiNet driver when it detects a new modem, but triggered in user-space instead of in the kernel. Being optional, the program can decide to fully skip doing it if it knows that the device is already initialized.

The steps that libqmi will perform while opening the QMI control port are specified in a bitmask of flags, and these also may include configuring the format of network packets to be transferred through the data network interface.

4.2.2 Client allocation and releasing

Once a QMI control device is open in a program using libqmi, it is then possible to request the allocation and releasing of new QMI clients (and therefore the *Client IDs*). This is achieved using the *CTL Client Allocation* and

[19] http://www.freedesktop.org/wiki/Software/libqmi
[20] http://www.freedesktop.org
[21] http://www.freedesktop.org/wiki/Software/ModemManager/
[22] https://projects.gnome.org/NetworkManager
[23] https://developer.gnome.org/glib
[24] https://developer.gnome.org/gobject
[25] https://developer.gnome.org/gio
[26] http://www.freedesktop.org/software/libqmi/libqmi-glib/latest

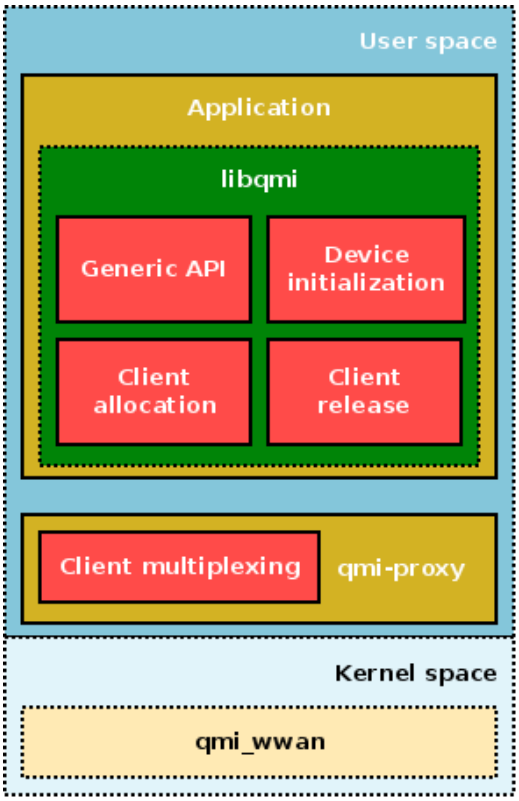

Figure 4: libqmi in user-space

CTL Client Release commands issued by the internal libqmi stack through the open character device. Note that as opposed to the GobiAPI behaviour, libqmi can handle multiple QMI clients for different services over the same open file descriptor.

libqmi can also benefit from the fact that the qmi_wwan driver doesn't automatically release *Client IDs* when the file descriptor gets closed. A set of flags when requesting to allocate or release a QMI client allow the program to re-use *Client IDs* that may have been left unreleased previously. The libqmi project offers the following tools:

- **qmicli:** libqmi comes with a handy command line interface tool which allows sending independent requests to the QMI device, also waiting for the reply of each request. A default run with qmicli will involve allocating and releasing a new QMI client for the service for which the action applies; but there is also the possibility to tell qmicli to re-use a specific *Client ID* and also to leave the *Client ID* unreleased when exiting.
- **qmi-network:** libqmi also comes with a bash script which internally uses qmicli. This script can setup a connection to the network using the QMI port from the commandline, and makes sure that the QMI client for the *WDS* service (the one providing the connection operations) is never released during the connection.

4.2.3 Client multiplexing

libqmi has an special **Proxy** flag that may be used when opening a device. If this flag is used, the process will spawn a child process running a '*qmi-proxy*' instance. This proxy process will setup an abstract Linux socket where it will listen to QMI messages from other processes. If all the programs which want to use a QMI port in the system use this same flag when respectively opening the device, they will all end up sharing a single qmi-proxy process which takes care of synchronizing the access to the /dev/cdc-wdmX character device. From another point of view, this qmi-proxy will be the only process really talking to the port.

So, even if the qmi_wwan driver doesn't allow multiple processes to use a single /dev/cdc-wdmX device (as

the GobiNet driver does), the multiplexing of different Clients from different processes can still be performed effectively in user-space.

4.3 Other user-space implementations using qmi_wwan

GobiAPI and libqmi are both general-purpose libraries making it easier to access the QMI port by other applications. But there are a couple of additional projects that also hold their own user-space implementations to access the port. It should be noted that none of the remaining implementations allow sharing access to the QMI port; i.e. once any of these opens the port to be used, no other application will be able to do so. This is of course a fully valid approach in e.g. embedded systems in which the Gobi modem may be used only for connection purposes.

- **oFono:** The oFono[27] telephony subsystem implements support for QMI communications with the qmi_wwan driver. It doesn't use any helper library to setup the communication with the modem, and handles the creation and parsing of the QMI messages itself.
- **uqmi:** The uqmi project[28], maintained by the OpenWRT[29] developers, re-uses the QMI message database implemented in libqmi, but generates code more suitable for embedded devices (e.g. not depending on GLib). Unlike libqmi, though, this project just creates a command line interface program with the most common commands to be run with the modem.

[27] https://ofono.org
[28] http://nbd.name/gitweb.cgi?p=uqmi.git
[29] https://openwrt.org

5 Resources

- **GobiNet and GobiAPI**
  - https://www.codeaurora.org/projects/all-active-projects/gobi-project
  - https://www.codeaurora.org/forums/gobi-mobile
  - https://www.codeaurora.org/patches/quic/gobi
- **qmi_wwan**
  - https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/net/usb/qmi_wwan.c
- **libqmi**
  - http://www.freedesktop.org/wiki/Software/libqmi
  - http://cgit.freedesktop.org/libqmi
  - http://lists.freedesktop.org/archives/libqmi-devel
  - http://sigquit.wordpress.com/2012/08/20/an-introduction-to-libqmi
  - http://sigquit.wordpress.com/2013/09/13/sharing-a-qmi-port-between-processes