Qualcomm
TECHNOLOGIES, INC

QMI Client API

Interface Specification

80-N1123-1 F

December 3, 2014

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures

Tables

Revision History

Revision	Date	Description
A	Apr 2010	Initial release.
B	Apr 2010	Made editing changes to conform to Qualcomm standards; no technical content was changed in this document revision.
C	Dec 2010	Updated to reflect the current state of the QMI Common Client Interface
D	Feb 2011	Changed parameter names in functions to increase the clarity of their purpose
E	Nov 2011	Numerous changes were made to this document. It should be read in its entirety.
F	Dec 2014	Added Chapters 3, 4, and 5.

1 Introduction

1.1 Purpose

This document explains the QMI Client APIs. These APIs can be used in conjunction with the autogenerated files from the QMI IDL compiler to write a client that can send messages to a service defined on the modem processor. This document reflects the latest APIs and behaviors of the QCCI Framework. Older releases might not have all APIs and behavior that are documented within.

1.2 Scope

This document is for customers who are familiar with the Qualcomm Messaging Interface (QMI) and who wish to develop a client that communicates with a QMI service.

The APIs mentioned in this document are subject to change based on further discussion within Qualcomm. However, a major change is not expected.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

Parameter types are indicated by arrows:

- → Designates an input parameter
- ← Designates an output parameter
- ↔ Designates a parameter used for both input and output

Shading indicates content that has been added or changed in this revision of the document.

1.4 References

Reference documents are listed in Table 1-1. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Table 1-1 Reference documents and standards

Ref.	Document	
Qualcomm Technologies		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1

1.5 Technical Assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://support.cdmatech.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

1.6 Acronyms

For definitions of terms and abbreviations, refer to [Q1]. Table 1-2 lists terms that are specific to this document.

Table 1-2 Acronyms

Acronym	Definition
QCCI	QMI Common Client Interface
QMI	Qualcomm messaging interface
SSR	subsystem restart
TCB	task control block
TLV	type-length-value

2 QMI APIs

The QMI APIs can be divided into five broad categories:

- Callback function prototypes
- Connection APIs
- Message sending APIs
- Release connection API
- Encode and decode APIs

2.1 Callback function prototypes

2.1.1 qmi_client_notify_cb()

This callback function is called by the QMI Common Client Interface (QCCI) infrastructure when a service event occurs, indicating that the service count has changed.

```
void  *qmi_client_notify_cb
(
  qmi_client_type                user_handle,
  qmi_idl_service_object_type    service_obj,
  qmi_client_notify_event_type   service_event,
  void                           *notify_cb_data
);
```

→	user_handle	Handle of the client
→	service_obj	Service object
→	service_event	Notifies the client whether the number of services (to which their service_obj relates) have increased or decreased. Values: ▪ QMI_CLIENT_SERVICE_COUNT_INC ▪ QMI_CLIENT_SERVICE_COUNT_DEC
→	notify_cb_data	User data passed in qmi_client_register_notify_cb

2.1.2 qmi_client_recv_raw_msg_async_cb()

This function is called by the QCCI infrastructure when a response is received after a request is sent using qmi_client_send_raw_msg_async(). Resp_buf is an encoded QMI message, and must be decoded by the client with the qmi_client_message_decode function.

Parameters

```
void  *qmi_client_recv_raw_msg_async_cb
(
  qmi_client_type                 user_handle,
  unsigned int                    msg_id,
  void                            *resp_buf,
  unsigned int                    resp_buf_len,
  void                            *resp_cb_data,
  qmi_client_error_type           transp_err
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	msg_id	Message ID
→	resp_buf	Pointer to the response
→	resp_buf_len	Length of the response
→	resp_cb_data	User data
→	transp_err	Error code; QMI_NO_ERR indicates success, otherwise indicates an issue receiving the message from the service.

2.1.3 qmi_client_recv_msg_async_cb()

This function is called by the QCCI infrastructure when a response is received after a request is sent using qmi_client_send_msg_async().

Parameters

```
void  *qmi_client_recv_msg_async_cb
(
  qmi_client_type                 user_handle,
  unsigned int                    msg_id,
  void                            *resp_c_struct,
  unsigned int                    resp_c_struct_len,
  void                            *resp_cb_data,
  qmi_client_error_type           transp_err
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	msg_id	Message ID
→	resp_c_struct	Pointer to the response
→	resp_c_struct_len	Length of the response
→	resp_cb_data	User data
→	transp_err	Error code; QMI_NO_ERR indicates success, otherwise indicates an error receiving or decoding the message from the service.

2.1.4 qmi_client_ind_cb()

This function is called by the QCCI infrastructure when an indication is received. This callback is registered at initialization. Ind_buf is an encoded QMI message, and must be decoded by the client with the qmi_client_message_decode function.

Parameters

```
void  *qmi_client_recv_raw_msg_async_cb
(
  qmi_client_type                 user_handle,
  unsigned int                    msg_id,
  void                            *ind_buf,
  unsigned int                    ind_buf_len,
  void                            *resp_cb_data
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	msg_id	Message ID
→	resp_c_struct	Pointer to the indication
→	resp_c_struct_len	Length of the indication
→	resp_cb_data	User data

2.1.5 qmi_client_error_cb()

This function is called by the QCCI infrastructure when the service terminates or deregisters. It is registered in the qmi_client_register_error_cb function. A client should call qmi_client_release() when the service is called, because their current handle to the service is no longer valid and they must re-initialize when the service comes back up.

Parameters

```
void  *qmi_client_error_cb
(
  qmi_client_type                 user_handle,
  qmi_client_error_type           error,
  void                            *err_cb_data
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	error	Error value
→	err_cb_data	User data

2.1.6 qmi_client_release_cb()

This function is called by the QCCI infrastructure when a connection has been fully released after calling qmi_client_release_async().

Parameters

```
void  *qmi_client_release_cb
(
  void                                *release_cb_data
);
```

→	err_cb_data	Cookie provided in qmi_client_release_async()

2.1.7 qmi_client_error_cb()

This callback function is called by the QCCI infrastructure when the service terminates or deregisters.

Parameters

```
void  *qmi_client_error_cb
(
  qmi_client_type                 user_handle,
  qmi_client_error_type           error,
  void                            *err_cb_data
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	error	Error code
→	err_cb_data	User data

2.2 Connection APIs

2.2.1 qmi_client_notifier_init()

This function is used for initializing a notifier with a service object. When a service that supports the service_obj arrives or exits the system, the signal or event object specified in os_params is set. The memory for os_params must be valid for the lifetime of the user_handle that is returned from the function. If os_params is declared on the stack, then user_handle must be freed via the qmi_client_release function before returning.

Parameters

```
extern  qmi_client_error_type
qmi_client_notifier_init
(
  qmi_idl_service_object_type     service_obj,
  qmi_client_os_params            *os_params,
  qmi_client_type                 *user_handle
);
```

→	service_obj	Service object
→	os_params	OS-specific parameters; can be a pointer to an event object, or signal mask and task control block (TCB)
←	user_handle	Handle used by the infrastructure to identify different clients

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR code – Otherwise

2.2.2 qmi_client_init()

This function is used for initializing a connection to a service. This function only provides the client with a valid user_handle that allows for communication with a service. The service is not notified of a client's existence until the first request message has been sent. The memory for os_params must be valid for the lifetime of the user_handle that is returned from the function. If os_params is declared on the stack, user_handle must be freed via the qmi_client_release function before returning.

Parameters

```
extern  qmi_client_error_type
qmi_client_init
(
  qmi_service_info                *service_info,
  qmi_idl_service_object_type     service_obj,
  qmi_client_ind_cb               ind_cb,
  void                            *ind_cb_data,
  qmi_client_os_params            *os_params,
  qmi_client_type                 *user_handle
);
```

→	service_info	Pointer to an entry in the service_info array returned by qmi_client_get_service_list()
→	service_obj	Service object
→	ind_cb	Indication callback function

→	ind_cb_data	Indication callback user data
→	os_params	OS-specific parameters; can be a pointer to an event object, or signal mask and TCB
←	user_handle	Handle used by the infrastructure to identify different clients

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR code – Otherwise

2.2.3 qmi_client_init_instance()

This is a blocking helper function that handles lookup and connection initialization to a service with a specific instance ID. This function internally calls qmi_client_get_service_list() and qmi_client_init() and might also create a notifier client in case the service is not already up. If a service of the required instance ID is not found, the function can block for a time longer than timeout before returning QMI_TIMEOUT_ERR.

Parameters

```
extern  qmi_client_error_type
qmi_client_init_instance
(
  qmi_idl_service_object_type       service_obj,
  qmi_service_instance              instance_id,
  qmi_client_ind_cb                 ind_cb,
  void                              *ind_cb_data,
  qmi_client_os_params              *os_params,
  uint32_t                          timeout,
  qmi_client_type                   *user_handle
);
```

→	service_obj	Service object
→	instance_id	Service instance
→	ind_cb	Indication callback function
→	ind_cb_data	Indication callback user data
→	os_params	OS-specific parameters; can be a pointer to an event object, or signal mask and TCB
→	timeout	Timeout in milliseconds; 0 = no timeout
←	user_handle	Handle used by the infrastructure to identify different clients

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR code – Otherwise

2.2.4 qmi_client_get_service_list()

Retrieves a list of services corresponding to the provided service object.

Parameters

```
qmi_client_error_type
qmi_client_get_service_list
(
  qmi_idl_service_object_type     service_obj,
  qmi_service_info                *service_info_array,
  uint32_t                        *num_entries,
  uint32_t                        *num_services
);
```

→	service_obj	Service object
←	service_info_array	Array to fill
↔	num_entries	Number of entries in the array as input; number of entries filled as output
←	num_services	Number of known services; if num_services > num_entries, a larger array is needed

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR – Otherwise

2.2.5 qmi_client_get_any_service()

Retrieves a single service corresponding to the provided service object. If multiple services exist in the system, it returns the first service found.

Parameters

```
qmi_client_error_type
qmi_client_get_any_service
(
  qmi_idl_service_object_type      service_obj,
  qmi_service_info                 *service_info
);
```

→	service_obj	Service object
←	service_info	Service information

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR – Otherwise

2.2.6 qmi_client_get_service_instance()

Retrieves a single service corresponding to the provided service object and specific instance ID.

Parameters

```
qmi_client_error_type
qmi_client_get_service_instance
(
  qmi_idl_service_object_type      service_obj,
  qmi_service_instance             instance_id,
  qmi_service_info                 *service_info
);
```

→	service_obj	Service object
→	instance_id	Instance ID of the service
←	service_info	Service information

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR – Otherwise

2.2.7 qmi_client_get_instance_id()

Obtains the instance ID for a specific service_info.

Parameters

```
qmi_client_error_type
qmi_client_get_instance_id
(
  qmi_service_info                *service_info,
  qmi_service_instance            *instance_id
);
```

→	service_info	Pointer to an entry in the service_info array
←	instance_id	Instance ID of the service_info entry

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR – Otherwise

2.2.8 qmi_client_register_error_cb()

Registers a callback that is called when a service terminates or deregisters.

Parameters

```
qmi_client_error_type
qmi_client_register_error_cb
(
  qmi_client_type                 user_handle,
  qmi_client_error_cb             err_cb,
  void                            *err_cb_data
);
```

→	user_handle	Opaque handle
→	err_cb	Pointer to callback function
→	err_cb_data	User data

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR – Otherwise

2.2.9 qmi_client_register_notify_cb()

Registers a callback that is called for service events. More details are contained in the qmi_client_notify_cb section.

Parameters

```
qmi_client_error_type
qmi_client_register_notify_cb
(
  qmi_client_type                user_handle,
  qmi_client_notify_cb           notify_cb,
  void                           *notify_cb_data
);
```

→	user_handle	Opaque handle
→	notify_cb	Pointer to callback function
→	notify_cb_data	User data

Return value

This function returns:

- QMI_NO_ERROR – Success
- ERROR – Otherwise

2.3 Message sending APIs

2.3.1 Asynchronous Messages

2.3.1.1 qmi_client_send_raw_msg_async()

Sends an asynchronous QMI service message. The caller is expected to encode the message before sending it through this function.

Parameters

```
extern qmi_client_error_type
qmi_client_send_raw_msg_async
(
  qmi_client_type                user_handle,
  unsigned int                   msg_id,
  void                           *req_buf,
  unsigned int                   req_buf_len,
  void                           *resp_buf,
  unsigned int                   resp_buf_len
  qmi_client_async_rsp_cb        *resp_cb,
  void                           *resp_cb_data
   qmi_txn_handle                *txn_handle
);
```

→	user_handle	Handle used by the infrastructure to identify the different clients
→	msg_id	Message ID
→	req_buf	Pointer to the request
→	req_buf_len	Length of the request
→	resp_buf	Pointer to where the response will be stored
→	resp_buf_len	Length of the response buffer
→	resp_cb	Callback function to handle the response
→	resp_cb_data	Callback user data
←	txn_handle	Handle used to identify the transaction

Return value

This function returns:

- QMI_NO_ERR – Sets transaction handle on success
- Error – Otherwise

2.3.1.2 qmi_client_send_msg_async()

Sends an asynchronous QMI service message. The function handles the encoding and decoding of the messages.

Parameters

```
extern qmi_client_error_type
qmi_client_send_msg_async
(
  qmi_client_type                user_handle,
  unsigned int                   msg_id,
  void                           *req_c_struct,
  unsigned int                   req_c_struct_len,
  void                           *resp_c_struct,
  unsigned int                   resp_c_struct_len
  qmi_client_recv_msg_async_cb   resp_cb,
  void                           *resp_cb_data
  qmi_txn_handle                 *txn_handle
);
```

→	user_handle	User handle
→	msg_id	Message ID
→	req_c_struct	Pointer to the request
→	req_c_struct_len	Length of the request
→	resp_c_struct	Pointer to where the response will be stored
→	resp_c_struct_len	Length of the response buffer
→	resp_cb	Callback function to handle the response
→	resp_cb_data	Callback user data
←	txn_handle	Handle used to identify the transaction

Return value

This function returns:

- QMI_NO_ERR – Sets the transaction handle on success
- Error – Otherwise

2.3.1.3 qmi_client_delete_async_txn()

Cancels an asynchronous transaction.

Parameters

```
extern qmi_client_error_type
qmi_client_delete_async_txn
(
  qmi_client_type   user_handle,
  qmi_txn_handle    async_txn_handle
);
```

→	user_handle	Client handle user handle
→	async_txn_handle	Sends handle async

Return value

This function returns:

- qmi_client_send_msg_async – async_txn_handle is returned by the function
- QMI_NO_ERR – Success
- Negative – Otherwise

Users should be aware of the potential race condition where an asynchronous response might be in the process of being handled by the users_rsp_cb callback up until this routine returns.

2.3.1.4 qmi_client_get_async_txn_id()

DEPRECATED. Gets a transaction ID from the transaction handle. This was added to support legacy QMI messages that require access to the transaction ID.

Parameters

```
extern qmi_client_error_type
qmi_client_get_async_txn_id
(
  qmi_client_type   user_handle,
  qmi_txn_handle    async_txn_handle,
  uint32_t          *txn_id
);
```

→	user_handle	Indicates a client handle user handle
→	async_txn_handle	Sends handle async
	txn_id	ID

Return value

This function returns:

- qmi_client_send_msg_async – async_txn_handle is returned by the function
- QMI_NO_ERR – Success
- Negative – Otherwise

Users should be aware of the potential race condition where an asynchronous response might be in the process of being handled by the users_rsp_cb callback up until this routine returns.

2.3.2 Synchronous Messages

2.3.2.1 qmi_client_send_raw_msg_sync()

Sends a synchronous QMI service message; it expects the user to encode the message before sending and decode the message after receiving.

Parameters

```
extern qmi_client_error_type
qmi_client_send_raw_msg_sync
(
  qmi_client_type                user_handle,
  unsigned int                   msg_id,
  void                           *req_buf,
  unsigned int                   req_buf_len,
  void                           *resp_buf,
  unsigned int                   resp_buf_len,
  unsigned int                   resp_buf_recv_len,
  unsigned int                   timeout_msecs
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	msg_id	Message ID
→	req_buf	Pointer to the request
→	req_buf_len	Length of the request
→	resp_buf	Pointer to where the response will be stored
→	resp_buf_len	Length of the response buffer
→	resp_buf_recv_len	Length of the response received
→	timeout_msecs	Timeout in milliseconds

Return value

This function returns:

- QMI_NO_ERROR – Success
- Error – Otherwise

2.3.2.2 qmi_client_send_msg_sync()

Sends a synchronous QMI service message; it provides the encoding/decoding functionality and the user gets the decoded data in the response structure provided.

Parameters

```
extern qmi_client_error_type
qmi_client_send_msg_sync
(
  qmi_client_type                user_handle,
  unsigned int                   msg_id,
  void                           *req_c_struct,
  unsigned int                   req_c_struct_len,
  void                           *resp_c_struct,
  unsigned int                   resp_c_struct_len,
  unsigned int                   resp_c_struct_recv_len,
  unsigned int                   timeout_msecs
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	msg_id	Message ID
→	req_c_struct	Pointer to the request
→	req_c_struct_len	Length of the request
→	resp_c_struct	Pointer to where the response will be stored
→	resp_c_struct_len	Length of the response buffer
→	resp_c_struct_recv_len	Length of the response received
→	timeout_msecs	Timeout in milliseconds

Return value

This function returns:

- QMI_NO_ERROR – Success
- Error – Otherwise

2.4 Release Connection API

Releases the connection.

Parameters

```
extern qmi_client_error_type
qmi_client_release
(
  qmi_client_type                  user_handle,
);
```

→	user_handle	Handle used by the infrastructure to identify different clients

Return value

This function returns:

- QMI_NO_ERROR – Success
- Error – Otherwise

2.5 Encode and Decode APIs

2.5.1 qmi_client_message_encode()

Encodes the body of a QMI message from the C data structure to the wire format.

Parameters

```
extern qmi_client_error_type
qmi_client_message_encode
(
  qmi_client_type                 user_handle,
  qmi_idl_type_of_message_type    req_resp_ind,
  unsigned int                    message_id,
  const void                      *p_src,
  unsigned int                    src_len,
  void                            *p_dst,
  unsigned int                    dst_len,
  unsigned int                    *dst_encoded_len
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	req_resp_ind	Type of message - Request, response, or indication
→	message_id	Message ID
→	p_src	Pointer to a C structure containing the message data
→	src_len	Length of the p_src C structure in bytes
←	p_dst	Pointer to the beginning of the first TLV in the message
→	dst_len	Length of p_dst buffer in bytes
←	dst_encoded_len	Pointer to the return value, the length of the encoded message

Return value

This function returns:

- QMI_NO_ERROR – Success
- Error – Otherwise

2.5.2 qmi_client_message_decode()

Decodes the body of a QMI message from the wire format to the C structure.

Parameters

```
extern qmi_client_error_type
qmi_client_message_decode
(
  qmi_client_type                user_handle,
  qmi_idl_type_of_message_type   req_resp_ind,
  unsigned int                   message_id,
  const void                     *p_src,
  unsigned int                   src_len,
  void                           *p_dst,
  unsigned int                   dst_len
);
```

→	user_handle	Handle used by the infrastructure to identify different clients
→	req_resp_ind	Type of message – Request, response, or indication
→	message_id	Message ID
→	p_src	Pointer to the beginning of the first TLV in the message
→	src_len	Length of the p_src buffer in bytes
←	p_dst	Pointer to the C structure for decoded data
→	dst_len	Length of the p_dst C structure in bytes

Return value

This function returns:

- QMI_NO_ERROR – Success
- Error – Otherwise

3 Use Cases

NOTE: This chapter was added to this document revision.

3.1 Usage Model

The usage model of the QCCI framework begins with the QMI client initializing a client handle. The client then sends QMI requests to the service either synchronously or asynchronously. In addition to the normal usage model, the client also handles a service restart or subsystem restart. Section 3.1.1 through Section 3.1.4 describe three use cases.

3.1.1 Initializing a Client

The following steps describe initializing a client.

1. The QMI client creates a notifier handle to receive service arrival and exit event notifications. Then the client either:
   a. Registers a notification callback with the notifier handle. If the specified QMI service is found, the notification callback is called immediately.
   b. Waits on a signal that was provided when the notifier handle was registered.
2. When the service arrives, the signal is set, and the notification callback is called to notify the client regarding the service arrival.
3. The QMI client resolves the service address after the service arrival notification is received.
4. The QMI client initializes a client handle to the service whose address is resolved at step 3. This client handle is then used to exchange QMI messages with the concerned QMI service.
5. When the client handle is initialized, an error callback function is registered with the client handle. This error callback is used to receive any notifications about the service restart or subsystem restart.
6. The QMI client can release the notifier handle at this point to avoid any resource leakage.
7. The QMI client sends the request messages either synchronously or asynchronously as explained in Section 3.1.2 and Section 3.1.3.
8. The QMI client releases the client handle when all the relevant QMI message communication is complete.

Figure 3-1 illustrates the basic call flow for various client events.

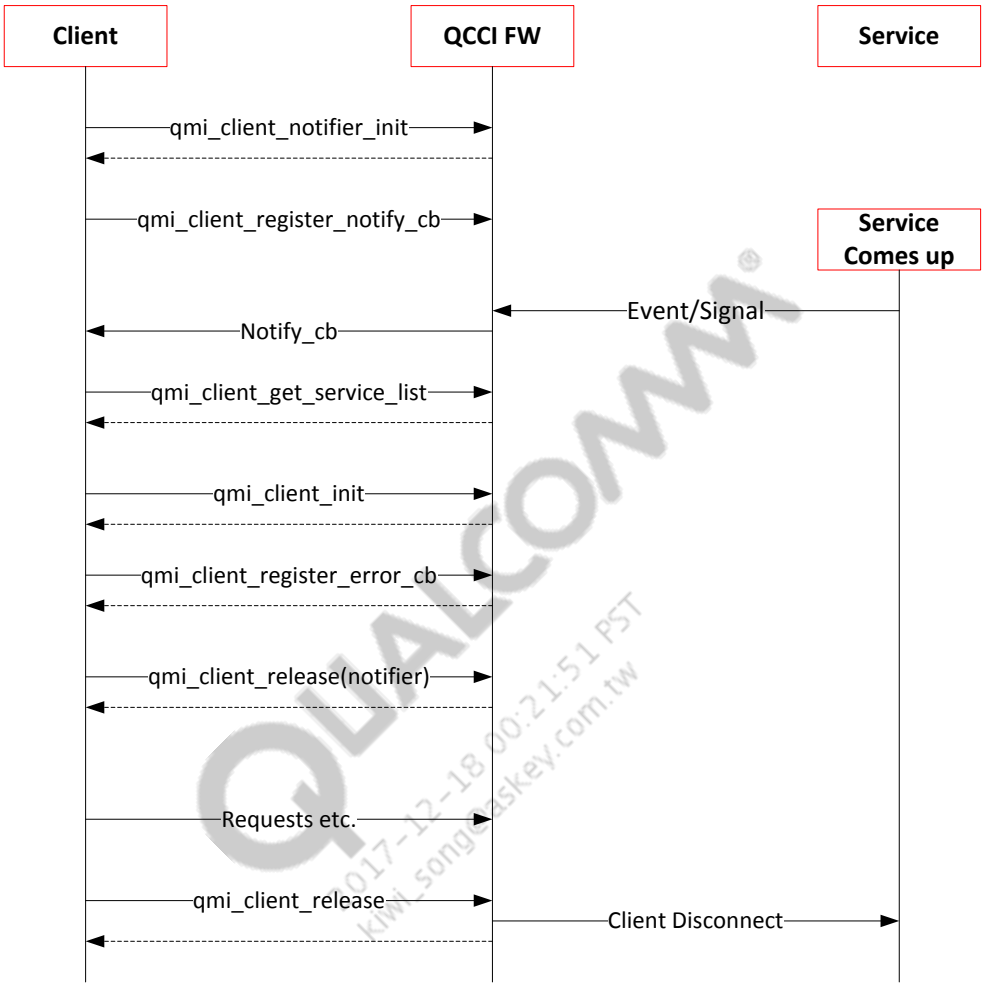

Figure 3-1 Initializing a client

3.1.2 Sending a Message Synchronously

Figure 3-2 illustrates the call flow for sending request messages synchronously.

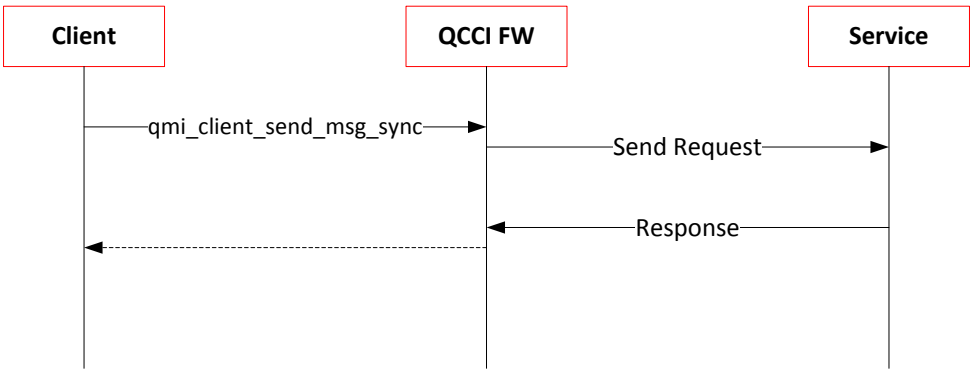

Figure 3-2 Sending a message synchronously

3.1.3 Sending a Message Asynchronously

Figure 3-3 illustrates the call flow for sending request messages asynchronously.

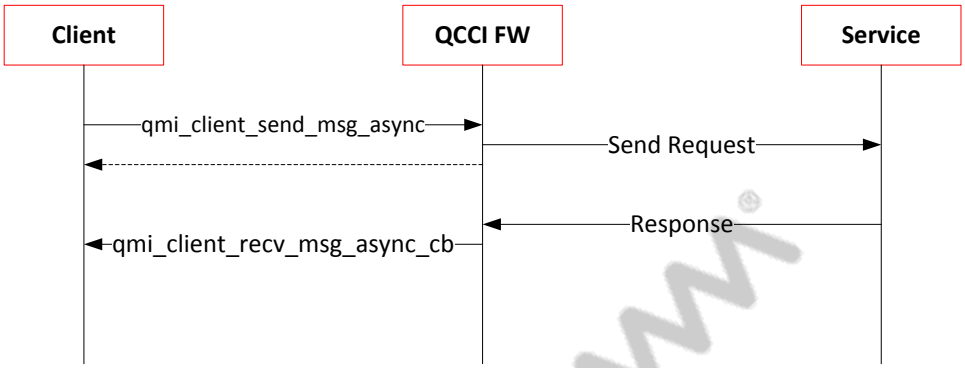

Figure 3-3 Sending a message asynchronously

3.1.4 Handling a Service or Subsystem Restart

The following steps describe handling a service or subsystem restart.

1. The QMI client receives an error callback that was registered along with the client handle when the specified QMI service restarts or the subsystem restarts.
2. At this point, the specified QMI client handle is marked as defunct and any attempt to send a QMI message through the defunct client handle results in a QMI_SERVICE_ERR error.
3. When the error callback is received, the QMI client releases the client handle.
4. The QMI client re-initializes the client handle as described in Section 3.1.1

Figure 3-4 illustrates the call flow for handling a service or subsystem restart.

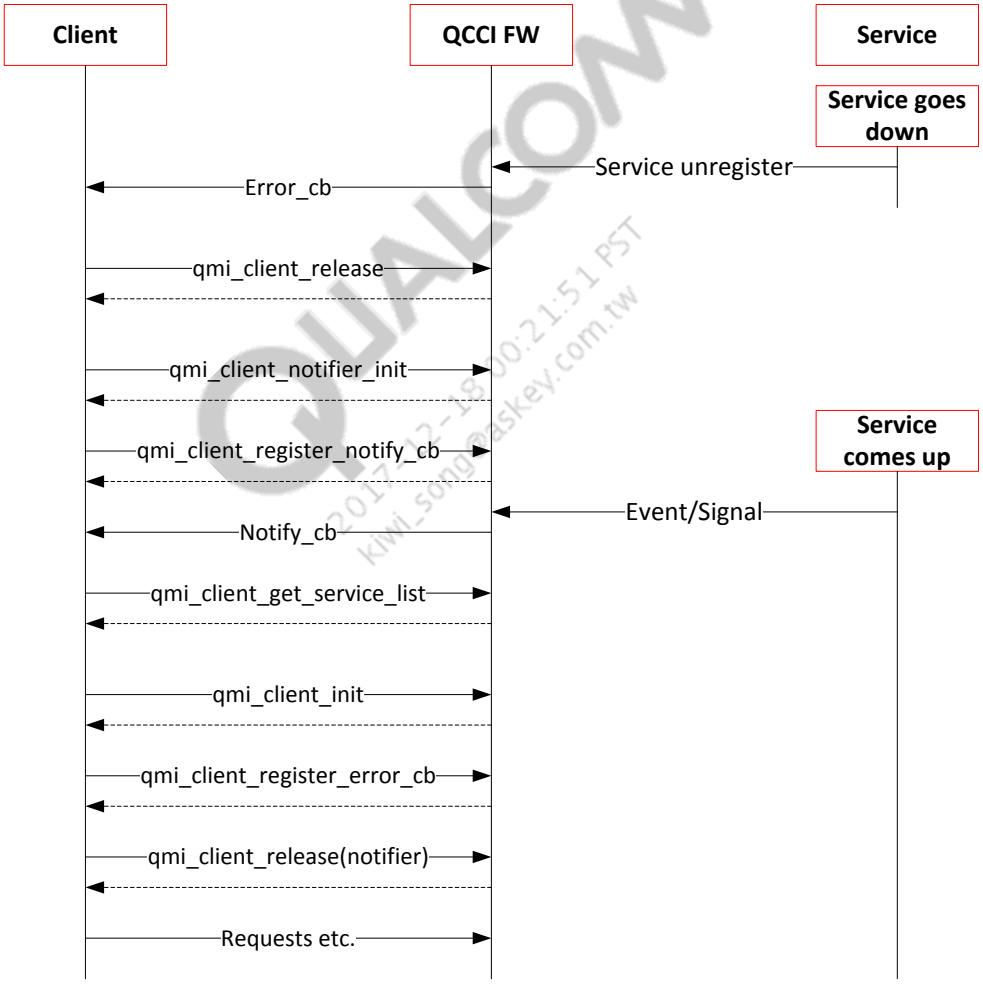

Figure 3-4 Handling a service/subsystem restart

4 Example Client Code

NOTE: This chapter was added to this document revision.

4.1 Initializing via qmi_client_init_instance

```
/* Create a client handle to the service.
   Pass in QMI_CLIENT_INSTANCE_ANY to connect to any service
   in the system and zero for the timeout to block the call
   until a service comes up to connect to. */
rc = qmi_client_init_instance(service_obj, QMI_CLIENT_INSTANCE_ANY,
indication_cb, ind_cb_data, &os_params, 0, &client_handle);
if (rc != QMI_NO_ERR)
{
  /* Handle the error. */
}
/* Register an error callback to handle SSR events. */
rc = qmi_client_register_err_cb(client_handle, err_cb, err_cb_data);
if (rc != QMI_NO_ERR)
{
  /* Handle the error. */
}
/* Connection is complete, messages can be sent to the
   service now using the client handle. Pass 0 for
   timeout to block indefinitely. */
rc = qmi_client_send_msg_sync(client_handle, msg_id, req_c_struct,
req_c_struct_len, resp_c_struct, resp_c_struct_len, 0);
```

4.2 Initializing via notifier_init()

```
/* Create a client handle to the service.
   Pass in QMI_CLIENT_INSTANCE_ANY to connect to any service
   in the system. */
rc = qmi_client_notifier_init(service_obj, os_params, &notifier_handle);
if (rc != QMI_NO_ERR)
{
  /* Handle the error. */
}
/* Wait on the signal provided in the os_params parameter of
   qmi_client_notifier_init (e.g., pthread_cond_wait). */
pthread_cond_wait(os_params->cond, os_params->mutex);
/* Wait over, look up service, getting the first service
   available in the service list. */
rc = qmi_client_get_any_service(service_obj, &service_info);
if (rc != QMI_NO_ERR)
{
  /* Handle Error */
}
rc = qmi_client_init(&service_info, service_obj, ind_cb, ind_cb_data,
os_params, &client_handle);
if (rc != QMI_NO_ERR)
{
  /* Handle Error */
}
/* No error in creating the client handle, delete the notifier
   handle, register an error callback to handle SSR events,
   and then begin sending messages. */
qmi_client_release(notifier_handle);
rc = qmi_client_register_err_cb(client_handle, err_cb, err_cb_data);
if (rc != QMI_NO_ERR)
{
  /* Handle the error. */
}
/* Connection is complete, messages can be sent to the
   service now using the client handle. Pass 0 for
   timeout to block indefinitely. */
```

```
rc = qmi_client_send_msg_sync(client_handle, msg_id, req_c_struct,
req_c_struct_len, resp_c_struct, resp_c_struct_len, 0);
```

4.3 Handling SSR

```
qmi_client_error_cb
(
  qmi_client_type                user_handle,
  qmi_client_error_type          error,
  void                           *err_cb_data
)
{
  /* Release the client handle and re-establish connection
     to the service, using the same method to create the
     initial connection. Do not do it in the error_cb, as
     it could result in deadlock. */
}
```

4.4 Handling Indications

```
qmi_client_ind_cb
(
  qmi_client_type                user_handle,
  unsigned int                   msg_id,
  void                           *ind_buf,
  unsigned int                   ind_buf_len,
  void                           *ind_cb_data
)
{
  /* Indications do not come decoded, decode the message manually */
  ind_msg_type ind_msg;
  int rc;

  rc = qmi_client_message_decode(user_handle, QMI_IDL_INDICATION, msg_id,
                                 ind_buf, ind_buf_len, &ind_msg,
                                 sizeof(ind_msg_type));
  /* If any lengthy processing must be done, the message should be passed
     to another task to avoid blocking the transport or reader task. */
```

5 OS-Specific Operations

NOTE: This chapter was added to this document revision.

This chapter describes the Operating System (OS) parameters argument to the various initialization methods, and the parameters' use and behavior in each OS. Each OS's port of QCCI must provide the following:

- qmi_cci_target_ext.h – Provides and abstracts all OS-specific behaviors from qmi_client.h
- qmi_client_os_params – Object definition that defines the OS parameters used by qmi_client.h
- QMI_CCI_OS_SIGNAL_WAIT – Macro for the client to use if it chooses to wait until a service event has been detected by the notifier
- QMI_CCI_OS_SIGNAL_TIMED_OUT – Macro to detect if a previous wait timed out

The following sections detail the behavior under each supported operating system.

5.1 Android, Linux, and QNX

Prototypes for Android™, Linux®, and QNX:

```
typedef struct {
    uint32_t sig_set;
    uint32_t timed_out;
    pthread_cond_t cond;
    pthread_mutex_t mutex;
} qmi_cci_os_signal_type;
typedef qmi_cci_os_signal_type qmi_client_os_params;
```

The OS parameters are the signaling object and a required argument whenever it is necessary for the user to explicitly wait for an event that could be set by the framework. The user is not required to edit or modify any of the members and the only requirement is to provide the storage of the signaling structure.

Function details are as follows:

- qmi_client_notifier_init() – The os_params parameter is required only if the user intends to wait for explicit events (server up or down) from the framework. The typical use case is a wait loop where the client waits for a service to come up. See Section 4.2 for example code. The client can provide NULL in the place of os_params in the call to qmi_client_notifier_init() and the framework refrains from signaling the client. This is typically used by clients that use the qmi_client_set_notify_cb() function.

- qmi_client_init() – The os_params parameter is not necessary and the user can provide NULL. The framework ignores any parameter provided as the os_params in this function call.
- qmi_client_init_instance() – The user is required to provide a valid pointer to the os_params parameter. The framework uses the storage internally to block the call if required.

5.2 Windows

The prototypes of the OS parameters change based on the location of the client (Kernel mode or User mode). Even though the prototypes differ, they have the same functional behavior and hence both are described together.

The OS parameters are the signaling object and a required argument whenever it is necessary for the user to explicitly wait for an event which could be set by the framework. The user is not required to edit or modify any of the members and the only requirement is to provide the storage of the signaling structure.

Function details are as follows:

- qmi_client_notifier_init() – The os_params parameter is required only if the user intends to wait for explicit events (server up or down) from the framework. The typical use case is a wait loop where the client waits for a service to come up. See Section 4.2 for example code. The client can provide NULL in the place of os_params in the call to qmi_client_notifier_init() and the framework refrains from signaling the client. This is typically used by clients that use the qmi_client_set_notify_cb() function.
- qmi_client_init() – The os_params parameter is not necessary and the user can provide NULL. The framework ignores any parameter provided as os_params in this function call.
- qmi_client_init_instance() – The user is required to provide a valid pointer to the os_params parameter. The framework uses the storage internally to block the call if required.

5.2.1 Kernel API

Prototypes for the Windows® Kernel mode API:

```
typedef struct {
    KEVENT event;
    BOOLEAN timed_out;
} qmi_cci_os_signal_type;
typedef qmi_cci_os_signal_type qmi_client_os_params;
```

5.2.2 User Mode API

Prototypes for the Windows User mode API:

```
typedef struct {
    HANDLE event;
    BOOLEAN timed_out;
} qmi_cci_os_signal_type;
typedef qmi_cci_os_signal_type qmi_client_os_params;
```

5.3 REX

The OS parameters contain the pointer to the TCB of the AMSS task and the signals that the client requires to be set upon events. The structure also provides storage to private members used for various activities.

Prototypes for REX:

```
typedef struct {
    /*************************
     *     USER SET MEMBERS    *
     *************************/
    rex_tcb_type  *tcb;
    rex_sigs_type  sig;
    rex_sigs_type  timer_sig;
    /*************************
     *     PRIVATE MEMBERS     *
     *************************/
    boolean        timer_inited;
    rex_timer_type timer;
    boolean        timed_out;
    boolean        inited;
} qmi_cci_os_signal_type;
typedef qmi_cci_os_signal_type qmi_client_os_params;
```

Function details are as follows:

- qmi_client_notifier_init() – The os_params parameter is required only if the user is required to wait for explicit events (server up or down) from the framework. The typical use case is in a wait loop where the client waits for a service to come up using the rex_wait function. See Section 4.2 for example code. The os_params parameter is a mandatory argument.
- qmi_client_init() and qmi_client_init_instance() – The os_params parameter is a mandatory argument but the framework does not store the pointer and hence it is not needed by the client to maintain the validity of the pointer to os_params with the scope of the handle. The framework sets the signal mask information provided to set the signal on the waiting task (in the case of synchronous message sending API calls). The user is responsible for ensuring that the signal mask is reserved in all threads that use the handle.

The client is required to set the following:

1. The TCB of the thread in which the client handle is used.
2. The signal mask reserved for events that might be used by the framework.
3. The signal mask reserved for timeouts that might be used by the framework. The timer_sig parameter might be the same as the signal mask. If the user provides 0 as the timer_sig, the framework refrains from initializing the timers associated with this handle.

5.4 QuRT

Prototypes for QuRT™:

```
typedef struct {
    /*************************
     *     USER SET MEMBERS    *
     *************************/
    qurt_anysignal_t  *ext_signal;
    unsigned int      sig;
    unsigned int      timer_sig;
    /*************************
     *     PRIVATE MEMBERS     *
     *************************/
    qurt_anysignal_t  signal;
    qurt_timer_t      timer;
    qurt_timer_attr_t timer_attr;
    boolean           timer_inited;
    boolean           timed_out;
} qmi_cci_os_signal_type;
typedef qmi_cci_os_signal_type qmi_client_os_params;
```

The user set members of the OS parameters are useful only for the qmi_client_notifier_init() or qmi_client_init_instance() functions. Otherwise, the framework uses internal signaling structures for blocking functions.

- ext_signal – If a valid pointer is provided, the framework sets the sig parameter signal mask on the provided signaling structure. The caller is responsible for the initialization of the signaling structure and the only use by the framework is in setting events on the signaling structure. The user can provide NULL, and in that case the framework initializes the private signal object and uses that for waiting.
- sig – The signaling mask that is set on the signaling object upon events.
- timer_sig – The signaling mask that is set by the timer upon timeout events.

Function details are as follows:

- qmi_client_notifier_init() – The os_params parameter is optional. If not provided, NULL is used and the framework does not signal the client. The primary use of not providing os_param is when the client chooses to use the notify callback qmi_client_set_notify_cb().
- qmi_client_init() – The os_params parameter is not used and is ignored. The client is expected to provide NULL in the place of os_params. The framework allocates signaling structures internal to the handle.
- qmi_client_init_instance() – The os_params parameter is mandatory and the framework uses the storage to initialize the signaling used while waiting until the service is up.