QMI Vendor-Specific Services with IDL/QCSI/QCCI

Application Note

80-N5706-1 E

May 31, 2013

Submit technical questions at:
https://support.cdmatech.com/

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

Contents

Figures

Tables

Revision History

Revision	Date	Description
A	Apr 2011	Initial release
B	May 2011	Changed document type on cover page
C	Aug 2011	Updated Chapter 3
D	Nov 2012	Added: ▪ Acronyms list ▪ Chapters 5 through 7 Updated the sample code in Section 2.3 and Section 2.4.
E	May 2013	Added Section 2.2

1 Introduction

1.1 Purpose

The Interface Description Language (IDL)/Qualcomm Messaging Interface (QMI) Common Service Interface (QCSI)/QMI Common Client Interface (QCCI) infrastructure allows licensees to add vendor-specific QMI services. Licensees can use IDL language and tools to define the messages and services and can use the QCSI APIs to write a service that can receive and respond to messages from a client as well as to send indication messages. Licensees can also use the QCCI APIs to write a client that can send and receive messages from a service, as well as to receive indication messages.

1.2 Scope

This document is prepared for external customers interested in creating QMI vendor-specific services with IDL/QCSI/QCCI APIs.

1.3 Conventions

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

Code variables appear in angle brackets, e.g., <number>.

Commands to be entered appear in a different font, e.g., **`copy a:*.* b:`**.

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 Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.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
Q2	*QCT API Guidelines and Tools*	80-N0846-1
Q3	*QMI Client API Reference Guide*	80-N1123-1
Q4	*QMI Common Service Interface API Reference Guide*	80-N1123-2
Q5	*Core Server Framework APIs Reference Guide*	80-N7262-1
Q6	*MDM9x15/MDM9x25 VSS QMI Service Application Note*	80-N5576-90

1.5 Technical Assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies 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
APQ	application-only processor – Qualcomm
HLOS	high-level operating system
IDL	interface description language
IPC	interprocessor communication
QCCI	QMI common client interface
QCSI	QMI common service interface
QMI	Qualcomm messaging interface
QSAP	QMI service access proxy
REX	Qualcomm real-time executive operating system
VSS	Virtual Switching System

2 Using IDL to Define a New Service

This chapter describes how to implement QMI vendor-specific services with IDL. For the QTI IDL language, refer to [Q2].

2.1 Maximum message size

A QMI message may not be more than 64K. IDL tools will output an error for any message whose maximum size exceeds 64K.

2.2 Maximum expected throughput

QMI is designed for control messaging only. QMI is only guaranteed to work reliably under the scenarios tested by Qualcomm. QMI depends on a reliable link layer, so any vendor extension that overloads the link layer can affect all modem control messaging. Attempting to use QMI to transport large files or large amounts of data can break all QMI control messaging for all QMI services. Please contact customer support if you have concerns.

2.3 Service ID assignment

Every vendor-specific service must have a service ID. Vendor-specific service IDs must be placed between QMUX_SERVICE_VENDOR_MIN and QMUX_SERVICE_VENDOR_MAX in the enum (defined in ds_qmi_svc_ext.h) to avoid using one that is already assigned for the existing services.

2.4 Sample code

In the example below, messages for a new service are defined. It is the vendor's responsibility to assign the appropriate service ID for a new service. In the example, it is set as QMUX_SERVICE_VENDOR_MIN +1.

Part I – Define all messages for the service. In this example, one message, one response, and one indication are defined.

```
/***************************************************************************
  @FILE     qmi_ping_api_v01.idl
  @BRIEF    Ping API for the QMI IDL
  @DESCRIPTION

  @COPYRIGHT Copyright (c) 2012 Qualcomm Technologies, Inc.
  All rights reserved.
  Confidential and Proprietary - Qualcomm Technologies, Inc.
***************************************************************************/
```

```
***************************************************************************/
include "common_v01.idl";
revision 0;
/** Maximum data size. */
const PING_MAX_DATA_SIZE = 65500;
//=========================================================================
/** @COMMAND QMI_PING
    @CMD_VERSION 1.0
    @BRIEF Tests the basic message passing between the client and service.
*/
//===========================================================================
//! @MSG     QMI_PING_REQ
//! @TYPE    Request
//! @SENDER  Control point
//---------------------------------------------------------------------------
message {
   //! Ping
   //! @VERSION_INTRODUCED 1.0
   //! @VERSION 1.0
   mandatory char ping[4];
   /**< Ping request. */
} ping_req_msg;
//===========================================================================
//! @MSG     QMI_PING_RESP
//! @TYPE    Response
//! @SENDER  Service
//---------------------------------------------------------------------------
message {
   //! Result Code
   //! @VERSION_INTRODUCED 1.0
   //! @VERSION 1.0
   mandatory qmi_response_type resp;      //!< Standard response type.
   /*   Standard response type. Contains the following data members:
        qmi_result_type - QMI_RESULT_SUCCESS or QMI_RESULT_FAILURE
        qmi_error_type  - Error code. Possible error code values are
                          described in the error codes section of
                          each message definition.
   */

   //! Pong
   //! @VERSION_INTRODUCED 1.0
   //! @VERSION 1.0
   optional char pong[4];
   /**< Pong response.
   */
```

```
} ping_resp_msg;

/** @ERROR
    QMI_ERR_NONE           No error in the request
    QMI_ERR_INTERNAL       Unexpected error occurred during processing
    QMI_ERR_MALFORMED_MSG  Message was not formulated correctly by the
                           control point, or the message was corrupted
                           during transmission
*/
//===========================================================================
//! @MSG     QMI_PING_IND
//! @TYPE    Indication
//! @SENDER  Service
//! @SCOPE   Per control point (unicast)
//---------------------------------------------------------------------------
message {
   //! Interruption
   //! @VERSION_INTRODUCED 1.0
   //! @VERSION 1.0
   mandatory char interruption[5];
   /**< Ping indication. */
} ping_ind_msg;

/** @DESCRIPTION
The request is used to…
The response message indicates…
The indication for this command indicates…
*/
```

Part II – List all the messages in the service and assign a service ID.

```
//======================================================================
// Service definition
//======================================================================
service ping {
   // <message ID>
   //! @ID QMI_PING
   ping_req_msg                QMI_PING_REQ,
   ping_resp_msg               QMI_PING_RESP,
   //! @ID QMI_QMI_PING_IND
   ping_ind_msg                QMI_PING_IND = 0x0020;
   } = QMUX_SERVICE_VENDOR_MIN +1;
```

2.5 IDL compile output

The IDL compiler outputs a .c file and a .h file. The .h file defines the data types the client must populate to send and receive data to and from the server. For IDL compiler operations, refer to [Q2].

Part I – C structure message definition

```
typedef struct {
   /* Mandatory */
   /* Ping */
   char ping[4];
   /**< Ping request. */
}ping_req_msg_v01;  /* Message */

typedef struct {
   /* Mandatory */
   /* Result Code */
qmi_response_type_v01 resp;
   /**<  Standard response type.  */

   /* Optional */
   /* Pong */
   char pong[4];
   /**< Pong response. */

}ping_resp_msg_v01;  /* Message */

typedef struct {
   /* Mandatory */
   /* Interruption */
   char interruption[5];
   /**< Oing indication. */
}ping_ind_msg_v01;  /* Message */
```

Part II – The generated Service Message ID

```
/* Service Message Definition */
#define QMI_PING_REQ_V01 0x0020
#define QMI_PING_RESP_V01 0x0020
#define QMI_PING_IND_V01 0x0020
```

Part III – The generated Service Object Accessor

```
/* Service Object Accessor */
/** This function is used internally by the autogenerated code. Clients
should use the macro ping_get_service_object_internal_v01( ) that takes in
no arguments. */
qmi_idl_service_object_type ping_get_service_object_internal_v01
   ( int32_t idl_maj_version, int32_t idl_min_version, int32_t
   library_version );

/** This macro should be used to get the service object */
#define ping_get_service_object_v01( ) \
   ping_get_service_object_internal_v01( \
        PING_V01_IDL_MAJOR_VERS, PING_V01_IDL_MINOR_VERS, \
        PING_V01_IDL_TOOL_VERS )
```

The .c file generated by the IDL tool is used by the QCCI/QCSI infrastructure encode/decode routines and is not used by the client directly.

2.5.1 Service_Object and Service_Object_Accessor

Service_Object and Service_Object_Accessor are generated by the IDL compiler in the output .c file and .h file. Service_Object contains meta information to encode/decode the messages. Service_Object_Accessor can be used to access Service_Object. Licensees are to register the service object to the QCSI and QCCI infrastructure. (See the registration function in Chapter 3 and client initialization in Chapter 4.) With Service_Object, the QCCI and QCSI are able to handle message encode/decode routines for the new service.

3 Creating a New Service with Core Server Framework

This chapter explains the core server framework and describes the process of creating a new service. The framework allows users to write to an object-based server to extend the core server object. The core server object is built on top of the QCSI with the primary purpose of hiding details and providing generic functionality that every service needs.

3.1 Objective

The basic objective of the core server framework is to make the process of writing the service easier and to provide a template that is easy to adopt.

Generic functionality offered by the framework is:

- Ability to create a task
- Automated delivery of indication messages to all the registered clients
- Ability to register a dispatch function table consisting of response functions that correspond to each message
- Ability to create various instances of the server object
- Ability to reuse the server code on various OS platforms

3.2 Internals

The core server object is modeled as shown in Figure 3-1. It uses task, dispatch, and container objects to provide all the generic functionality. The new server generally extends the core server object to inherit all the generic functionality. Details of extending the core object are described in Section 3.3 with code examples. The server also extends the indication object to take care of the indications.

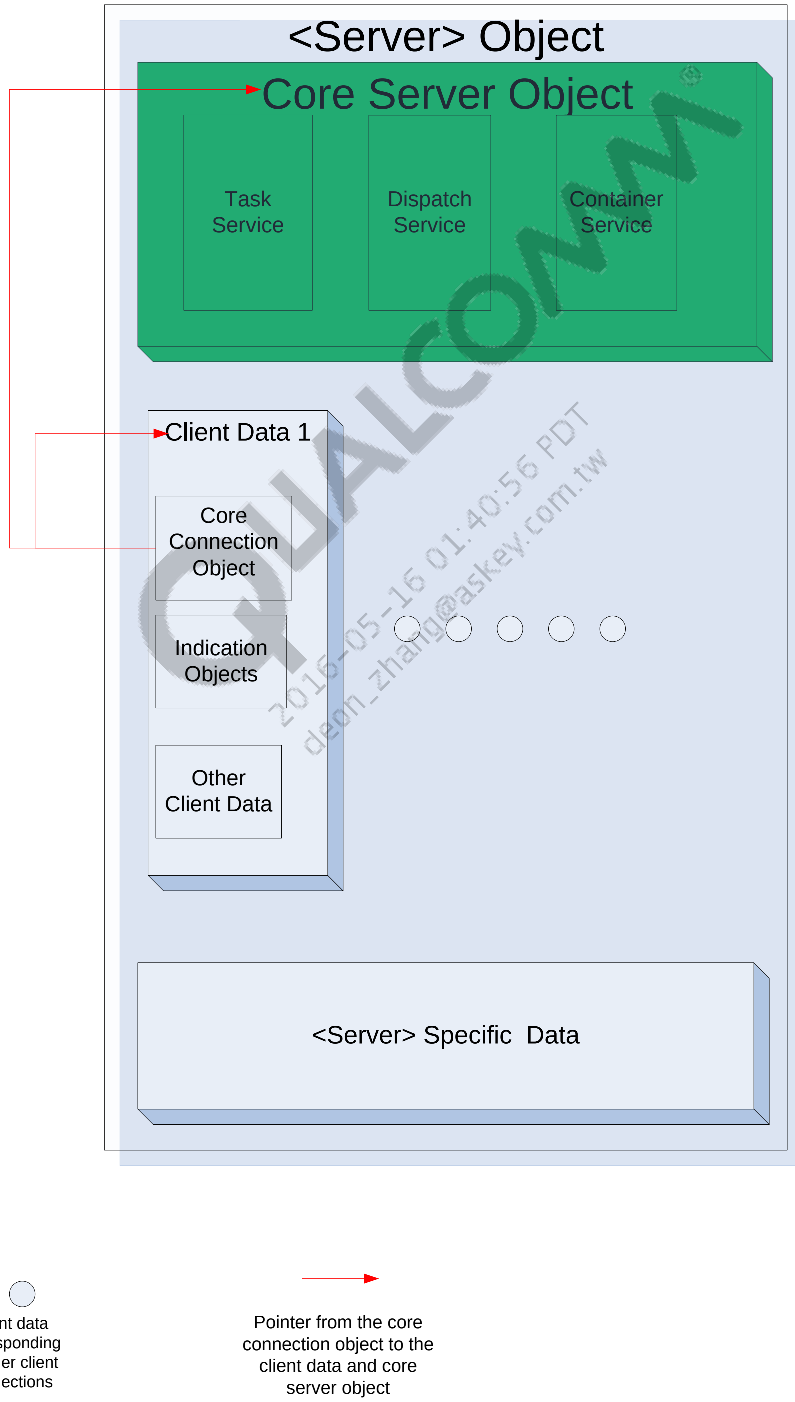

Figure 3-1 Core server object

In Figure 3-1, the arrow represents a pointer inside the core connection object that points to client data and core server data. The core connection objects are stored in the container object. This facilitates the core server framework to handle the indication messages on behalf of the server.

The task service, dispatch service, and container service are part of the core server object and are described as follows:

- Task service – This service is responsible for creating a task on behalf of the server. The processing of the incoming messages occurs in the context of this task. The default priority of the task created is one greater than the DIAG task.
- Dispatch service – The dispatch service provides the capability to register a dispatch table that consists of the response functions. Details of the dispatch table and response functions are described in Section 3.3.3.
- Container service – This service provides a storage mechanism that is used by the core server framework to store the connection objects.

3.3 Extending core server framework

Extending the core server framework primarily involves including the core server object as the first element of the server object and using the core server framework APIs to write the extended server. The following sections are to be read in conjunction with the sample source and header file provided. The sample code is for a ping service.

3.3.1 Extending core server object

The following sample code below shows how to extend the core server object:

```
/* This is the server object */
typedef struct ping_server_class {
    /* Core object should be first element in this struct */
    qmi_core_server_object_type       core_object;
    /* Client data */
    ping_server_client_data_type      client_data[MAX_NUM_CLIENTS];
    uint32_t                          client_index; /* Index into the
                                      client data array */
    uint32_t                          client_counter; /* Counts number of
                                      active clients */
    /* Rest of the server data goes here */
}ping_server_class_type;

/* Public APIs */

/* Ping server object */
typedef ping_server_class_type ping_server;
```

The core server object is essentially extended by using it as the first element of the structure. The ping_server_client_data_type element holds all the data structures required to track the client that connects through connect callback. Define the ping_server_client_data_type element as shown in the following sample code:

```
/* This struct keeps track of the clients that connect through connect
callback */
typedef struct ping_server_client data {
   /* First element of structure should be connection object */
   qmi_core_conn_obj_type        conn_obj;
   /* Indications should be defined here, probably with descriptive names*/
   ping_indication_data_type     ping_ind;
   /* Rest of the client data goes here */
   uint32_t                      transaction_counter; /* Counts number of
                                 transactions served */
   uint8_t                       active_flag;
}ping_server_client_data_type;
```

The first element in this structure must be the core connection object type (qmi_core_conn_obj_type) that allows the framework to be aware of indications defined by the server and clients connected to the server. The core connection object is followed by indication objects that might interest a client. Details on indications are described in Section 3.3.4.

The server implementation then needs to provide a constructor function to initialize its data and the core server object. The sample code is as follows:

```
ping_server_error
ping_server_init(ping_server         *me,
                 char                *object_name,
                 uint32_t            *instance_id)
{
    qmi_core_server_error_type rc;
    uint32_t index;
    uint32_t priority = 14;
    unsigned long sig – QMI_PING_SVC_WAIT_SIG;

    /* Construct and Initialize the core server object */
    rc = qmi_core_server_new(&(me->core_object),
                            object_name,
                            instance_id, /* Instance id */
                            1, /* Task flag */
                            (void*)ping_server_event_loop,
                            (void*)&priority,
                            NULL,
                            (void *)&sig,
                            NULL,
                            ping_server_dispatcher,
                            sizeof(ping_server_dispatcher)/
                            sizeof(qmi_msg_handler_type));

    if (rc == QMI_CORE_SERVER_NO_ERR )
    {
        /* Initialize the client data */
        for (index = 0; index < MAX_NUM_CLIENTS; index++ ) {
             memset(&(me->client_data[index].conn.obj),0,sizeof
             (qmi_core_conn_obj_type));
             /* Initialize each indication object declared in the client
             structure */
             ping_server_initialize_indication_data
             (&(me->client_data[index].ping_ind));

             me->client_data[index].transaction_counter = 0;
             me->client_data[index].active_flag = 0;
        }
        me->client_counter = 0;
        me->client_index = 0;
        /* Initialize rest of the server data here */
    }
    return rc;
}
```

The core server object is initialized by the API provided by the core server framework APIs. The second-to-last argument of the qmi_core_server_new API takes a dispatch table that consists of the message handler functions. The server implementation must also provide the destructor function to destruct the server object.

3.3.2 Server and client registration

The server also implements a server registration as illustrated in the following sample code:

```
ping_server_error
ping_server_register(ping_server *me )
{

    qmi_core_server_error_type rc;

    if ( qmi_core_server_check_valid_object(me) != QMI_CORE_SERVER_NO_ERR )
           return PING_SERVER_INVALID_OBJECT;

    /* Registering the service object and callbacks with QCSI framework
       using the core server object */

    rc = qmi_core_server_register(me,
                                  ping_get_service_object_v01( ),
                                  (qmi_csi_connect)ping_server_connect,
                                  (qmi_csi_disconnect)ping_server_disconnect,
                                  (qmi_csi_process_req)ping_server_
                                  process_req);
    return rc;
}
```

As shown in the code sample, the server must implement the callbacks using the core server framework utility APIs. The callbacks are called with the server object as the server object is passed in during the registration process. In the implementation of ping_server_connect callback, the client information gets registered with the core server framework. The ping_server_process_req callback dispatches the incoming message to the appropriate response function registered via the dispatch table. The dispatch table was registered during the creation of the server object. The deregistration of clients occurs when ping_server_disconnect is called from the framework. The server implementation must also have a server deregistration function. Details of the implementation are shown in the code supplied.

3.3.3 Response/request handler functions and dispatch table

The server implementation provides the message handler functions to each message request. These handler functions can then be registered via a dispatch table to the core server framework. A sample handler function and dispatch table are shown in the following code sample:

```
ping_server_error
qmi_ping_data_req_v01_handler(void             *server_data,
                          void             *conn_obj,
                          qmi_req_handle      req_handle,
                          uint32_t            msg_id,
                          void             *req_c_struct,
                          uint32_t            req_c_struct_len)
{

   ping_data_req_msg_v01  *recv_msg;
   ping_data_resp_msg_v01 *resp_msg;
   uint32_t index;

   recv_msg = (ping_data_req_msg_v01 *)req_c_struct;
   resp_msg.data_len = recv_msg->data_len;

   /* At this point, the server can do some relevant work based on incoming
      message request. The incoming message will be in req_c_struct
      argument. */

   /* In the case of incoming message request QMI_PING_DATA_REQ_V01 the
      relevant work is to simply add one to each element of the data array.
      */
   for (index = 0;index < recv_msg->data_len;index++)
        resp_msg.data[index] = recv_msg->data[index] + 1;

   resp_msg.resp.error = 0;
   resp_msg.resp.result = 0;

   /* Send a response back to client */
   return qmi_core_server_send_resp(req_handle,
                                    QMI_PING_DATA_RESP_V01,
                                    &resp_msg,
                                    sizeof(ping_data_resp_msg_v01));
}
```

```
/*=========================================================================
 Dispatch table for ping server.
=========================================================================*/
qmi_msg_handler_type ping_server_dispatcher[NUM_OF_MSGS] = {
     {QMI_PING_REQ_V01,(qmi_dispatch_fn_type)qmi_ping_req_v01_handler},
     {QMI_PING_DATA_REQ_V01,(qmi_dispatch_fn_type)qmi_ping_data_req_v01_
     handler},
     {QMI_PING_DATA_IND_REG_REQ_V01,(qmi_dispatch_fn_type)qmi_ping_data_
     ind_reg_req_v01_handler},
};
```

The server implementation gets the required information from the clients in the req_c_struct and msg_id arguments. The implementation processes this information and responds to the client with the appropriate response message.

3.3.4 Handling indications

The core server framework provides indication objects that can be extended by the server implementation to exercise the services provided by the indication service. The indication service provides basic functionality to send, initialize, activate, and deactivate the indications. Indication objects are of two types:

- One-shot indication – This is the use case where the client must register for indication and, when an event gets triggered (on the server side), the client gets the registered indication. The client must reregister the indication each time it wants any further indication of the given type.
- Unicast indication – This is the use case where the client gets unsolicited indications from the server

The server implementation initializes the indication objects as one of the above types.

3.3.4.1 Extending indication object

The following sample code shows how to extend the indication object:

```
/* This structure keeps track of the data we receive from the clients via
   ping_data_ind_reg_req_msg_v01 msg. This data is then used to deliver
   indications to clients via ping_data_ind_msg_v01 */
typedef struct ping_indication_data {
    qmi_indication_type base_ind;
    uint16_t            num_inds;
    uint16_t            ind_size;
    uint16_t            ind_delay;
    uint16_t            num_reqs;
}ping_indication_data_type;
```

The structure must have qmi_indication_type as the first member. The struct members hold the registration information received from the clients as a registration message. The registration information is then used by the server implementation to send indications to the clients.

3.3.4.2 Indication initialization

The extended indication object is initialized by the server implementation during the creation of the server object. The following sample code illustrates the indication object initialization:

```
ping_server_error
ping_server_initialize_indication_data(ping_indication_data_type *ind)
{
    ping_server_error rc;

    if (ind == NULL )
        return PING_SERVER_INVALID_OBJECT;

    memset(ind,0,sizeof(ping_indication_data_type));

    rc = qmi_indication_initialize((qmi_indication_type *)ind,
                                   QMI_UNICAST_IND,
                                   QMI_PING_DATA_IND_V01,
                                   sizeof(ping_indication_data_type),
                                  (qmi_indication_send_fn_type)
                                   ping_server_send_data_ind );

    /* Initialize any other extended indication data here */
    return rc;
}
```

The last argument to the API qmi_indication_initialize takes a callback function that is called when server implementation tries to send the indication based on an event. This function is called automatically by the framework for all registered clients of the server. If the indication is a one-shot type, the server implementation must activate the indication when it gets the registration request for the indication. The API qmi_indication_set_flag is used to activate the indication.

3.3.4.3 Sending indications

The server implementation must have a function call that actually sends the indication based on an event specific to the server. The following sample code illustrates the function:

```
ping_server_error
ping_server_send_ind( ping_server    *me,
                      int32_t        msg_id)
{
    ping_data_ind_msg_v01  ind_msg;
    ping_server_error      rc;

    switch(msg_id)
    {
    case   QMI_PING_DATA_IND_V01;
         rc = qmi_core_server_send_ind(&(me->core_object),
                                       msg_id,
                                       &ind_msg,
                                       sizeof(ping_data_ind_msg_v01));
         break;

    default;
         rc = PING_SERVER_UNKNOWN_MESSAGE;
         break;
    }
   return rc;
}
```

The framework calls the callback function (ping_server_send_data_ind) registered during the indication initialization on each client known to the server.

3.3.5 Platform-specific code

The core server framework allows the server implementation to be portable across different HLOS platforms. The server implementation must have a platform-specific file that consists of the OS-specific implementation of the event loop as illustrated in the following sample code:

```
void ping_server_event_loop(dword param)
{
   rex_sigs_type wait_on,sig_received;
   qmi_csi_os_params         *os_params;

   ping_server     *ping_srv_obj;
   ping_srv_obj =  (ping_server *)param;
   os_params = ping_server_get_os_params(ping_srv_obj);
   wait_on = QMI_PING_SVC_WAIT_SIG;

   while (1) {
       sig_received = rex_wait(wait_on);
       rex_clr_sigs(os_params->tcb,sig_received);
       ping_server_handle_event(ping_srv_obj,
                                sig_received);
   }
}
```

The preceding code is the implementation of the ping server on REX. The event is triggered when a message is received from the underlying transport. The ping_server_handle_event function is implemented as follows:

```
void ping_server_handle_event(ping_server *me,
                              void       *event)
{
    /* Call the process event function. */
    /* Server context at this point knows that there is an event. */
    /* The event is a rex signal(in case of rex implementation ) that can
       be used by the server. */
    /* qmi_core_server_handle_event will call qmi_csi_handle_event function
       and that will eventually call the ping_server_process_req callback
       */
    qmi_core_server_handle_event(me);
}
```

This function call ultimately results in the ping_server_process_req callback getting called. The callback implementation then dispatchs the received message to the appropriate response/request handler function registered initially with the core server framework. The event parameter passed in the above function is not used in this example but can be used by the server as required. The callback implementation is as follows:

```
qmi_csi_cb error
ping_server_process_req(void              *connection_handle,
                        qmi_req_handle    req_handle,
                        int               msg_id,
                        void              *req_c_struct,
                        int               req_c_struct_len,
                        ping_server       *me)
{

   qmi_core_server_error_type core_server_err;
   qmi_core_conn_obj_type    *conn_obj;

   DEBUG("PROCESS REQUEST CALLBACK CALLED..........\n",0,0,0);

   if  (qmi_core_server_check_valid_object(me) != QMI_CORE_SERVER_NO_ERR ||
        connection_handle == NULL )
        return QMI_CSI_SB_INTERNAL_ERR;

   conn_obj = (qmi_core_conn_obj_type *)connection_handle;

   /* Response/Request handler functions will eventually get called based
      on msg_id received */

   core_server_err = qmi_core_server_dispatch_msg(conn_obj,
                                                  me,
                                                  req_handle,
                                                  msg_id,
                                                  req_c_struct,
                                                  req_c_struct_len);

   if (core_server_err != QMI_CORE_SERVER_NO_ERR )
       return QMI_CSI_CB_INTERNAL_ERR;
   else
       return QMI_CSI_CB_NO_ERR;
}
```

4 Creating a Client with QCCI APIs

This chapter describes how to create a client with QCCI APIs.

The functions used are:

- qmi_client_notifier_init – This function is used for initializing a notifier with a service object. When a new service is registered to support the service object, the signal or event object specified in os_params is set.
- qmi_client_get_service_list – This function is used to get addressing information for accessing the service. If a service that matches the type in the service_obj parameter is available, the available_service_info_array parameter is filled in with information that will be passed to the qmi_client_init function.
- qmi_client_init – This function is used for initializing a connection to a service.

Figure 4-1 shows the sequence for creating a client with QCCI APIs.

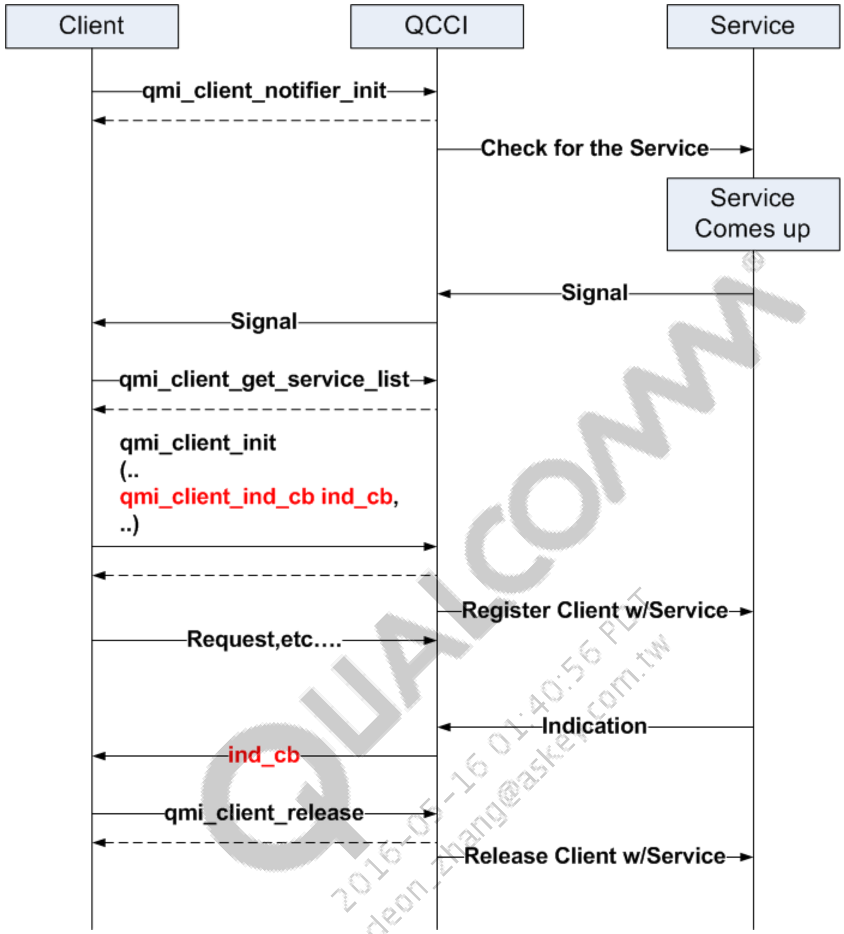

Figure 4-1 Creating a client with QCCI APIs

4.1 Sample code for client initialization

The following code is an example of client initialization and message delivery.

```
void qmi_ping_client_thread(uint32 handle)
{
  qmi_client_type clnt, notifier;
  qmi_txn_handle txn;
  uint32_t num_services, num_entries=10, i=0, num_services_old=0;
  int rc;
  qmi_cci_os_signal_type os_params;
  qmi_service_info info[10];
  qmi_idl_service_object_type ping_service_object =
ping_get_service_object_v01();

  os_params.tcb = rex_self();
  os_params.sig = QMI_CLNT_WAIT_SIG;
```

```
    os_params.timer_sig = QMI_CLNT_TIMER_SIG;

    if (!ping_service_object)
    {
      MSG_HIGH("PING: ping_get_service_object failed, verify
  qmi_ping_api_v01.h and .c match.\n",0,0,0);
    }

    rc = qmi_client_notifier_init(ping_service_object, &os_params,
  &notifier);

    /* Check if the service is up, if not wait on a signal */
    while(1)
    {
      QMI_CCI_OS_SIGNAL_WAIT(&os_params, 0);
      QMI_CCI_OS_SIGNAL_CLEAR(&os_params);

      /* The server has come up, store the information in info variable */
      num_entries=10;
      rc = qmi_client_get_service_list( ping_service_object, info,
  &num_entries, &num_services);
      MSG_HIGH("PING: qmi_client_get_service_list() returned %d num_entries =
  %d num_services = %d\n", rc, num_entries, num_services);

      if(rc != QMI_NO_ERR || num_services == num_services_old)
        continue;

      MSG_HIGH("PING: new service(s) discovered! num_services_old=%d
  num_services=%d\n", num_services_old, num_services, 0);

      num_services_old = num_services;

      for(i = 0; i < num_services; i++)
      {
        rc = qmi_client_init(&info[i], ping_service_object, ping_ind_cb,
  NULL, &os_params, &clnt);

        MSG_HIGH("PING: qmi_client_init[%d] returned %d\n", i, rc, 0);

        ping_basic_test(&clnt,&txn,1);
        ping_basic_test(&clnt,&txn,10);

        rc = qmi_client_release(clnt);
        MSG_HIGH("PING: qmi_client_release[%d] returned %d\n", i, rc, 0);
      }
```

```
  }
  /* Not reached */
  // rc = qmi_client_release(notifier);
  // MSG_HIGH("PING: qmi_client_release notifier returned %d\n", rc, 0,0);
}

void qmi_ping_client_start(void)
{
  strncpy( qmi_ping_client_tcb.task_name,
      "QMI_PING_CLNT",
      REX_TASK_NAME_LEN + 1 );

  rex_def_task( &qmi_ping_client_tcb,
      &qmi_ping_client_stack,
      QMI_PING_CLIENT_STACK_SIZE,
      10,
      qmi_ping_client_thread,
      0 );
}
```

The following example shows how to send a message to the service.

```
/*===========================================================================
  FUNCTION ping_basic_test
===========================================================================*/
/*!
@brief
  This function sends a number of basic ping messages asynchronously

@param[in]   clnt                Client handle needed to send messages

@param[in]   txn                 Transaction handle

@param[in]   num_pings           Number of pings to send

*/
/*===========================================================================
*/
void ping_basic_test
(
 qmi_client_type *clnt,
 qmi_txn_handle *txn,
 int num_pings
 )
{
  int i,rc;
  ping_req_msg_v01 req;
  ping_resp_msg_v01 resp;
```

```
    /* Set the value of the basic ping request */
    memcpy(&req, "ping", 4);
    MSG_HIGH("PING: Basic Ping Test with %d async ping
  messages.\n",num_pings,0,0);
    for (i=0;i<num_pings;++i)
    {
        rc = qmi_client_send_msg_sync(*clnt, QMI_PING_REQ_V01, &req,
  sizeof(req),
            &resp, sizeof(resp), 0);
        MSG_HIGH("PING: qmi_client_send_msg_sync returned %d on loop %d\n",
  rc,i,0);
        if (rc != 0){
          MSG_HIGH("PING: send_msg_sync error: %d\n",rc,0,0);
        }
        else
        {
          MSG_HIGH("PING: Pong Response received\n",0,0,0);
        }
    }
  }
```

4.2 Vendor-implemented callback functions

Qualcomm Technologies, Inc. defines the following callback function prototypes; vendors are to implement these callback functions:

- qmi_client_recv_msg_async_cb
- qmi_client_recv_raw_msg_async_cb
- qmi_client_ind_cb

4.2.1 qmi_client_recv_msg_async_cb

This callback function is called by the QCCI infrastructure when a response is received after a request is sent using qmi_client_send_msg_async(). Figure 4-2 is an example of how the callback function is passed to the QCCI infrastructure and how the callback function is called by the QCCI infrastructure.

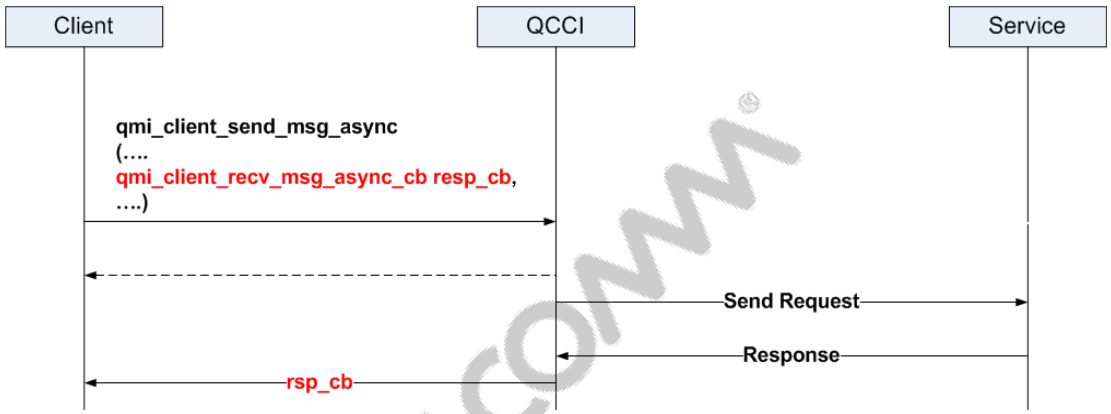

Figure 4-2 qmi_client_recv_msg_async_cb callback function

The following is an example of the callback function:

```
/*=========================================================
  CALLBACK FUNCTION ping_rx_cb
=========================================================*/
/*!
@brief
  This callback function is called by the QCCI infrastructure when
  infrastructure receives an asynchronous response for this client

@param[in]   user_handle         Opaque handle used by the infrastructure
                                 to identify different services.

@param[in]   msg_id              Message ID of the response

@param[in]   buf                 Buffer holding the decoded response

@param[in]   len                 Length of the decoded response

@param[in]   resp_cb_data        Cookie value supplied by the client

@param[in]   transp_err          Error value

*/
/*===========================================================*/
static void ping_rx_cb
```

```
(
 qmi_client_type              user_handle,
 unsigned long                msg_id,
 void                              *buf,
 int                                 len,
 void                             *resp_cb_data,
 qmi_client_error_type        transp_err
 )
{
  --pending_async;
  /* Print the appropriate message based on the message ID */
  switch (msg_id)
  {
    case QMI_PING_RESP_V01:
      MSG_HIGH("PING: Async Ping Response received\n",0,0,0);
      break;
  default:
      break;
  }
}
```

5 Modem Restart

The QCCI and QCSI are connectionless, but the Interprocessor Communication (IPC) router supports notification for end-point termination. Each server is an end-point, as is each client; so when a client closes its handle, the server is notified, and vice versa. When a processor goes down, all the nodes on the network are notified of the end-points on that processor.

5.1 QCCI subsystem restart

From the client perspective, there are two ways to determine that the server being talked to has gone down:

1. All qmi_client_send_* APIs return a negative error code, showing an error while writing to the transport (QMI_SERVICE_ERR)
   - All sync send message calls unblock (qmi_client_send_msg_sync and qmi_client_send_raw_msg_sync) and return QMI_SERVICE_ERR
   - All async calls (qmi_client_send_msg_async and qmi_client_send_raw_msg_async) that have not received a response have their callback called with the error set to QMI_SERVICE_ERR
2. The client can register an error notification callback via qmi_client_register_error_cb() after calling qmi_client_init(). This enables passive clients (e.g., clients that register and wait for indications) to be notified of the server going down. If the server goes down before the error callback is registered, the callback is invoked right away in the client's thread context. Since the communication model is connection-less, qmi_client_init() can still succeed if the server goes down before the client is created. An extra verification has been added to make sure the server exists before qmi_client_init() can return a success response.

   Once the client is aware of the server going down, it must reinitialize by doing the following:

   - Close the existing handle by calling qmi_client_release
   - Call one of the qmi_client_get_* functions (in conjunction with qmi_client_notifier_init() if the notifier has been released) to obtain the new address of the server

Call qmi_client_init() to obtain a new handle

6 Service Object Inheritance

Service Object Inheritance is a feature introduced in major version 5 of the IDL compiler and encode/decode library. This feature allows a client and server to have a child service object inherit all of the messages and TLVs of the parent service object. A child service object can also override a parent's messages, replacing them with its own version of a specific message.

This capability is useful for allowing a service to separate its implementation of some private or restricted interfaces from the rest of the service. For restricted interfaces (e.g., customer-specific interfaces) a child service can be implemented in a library that is not always linked into the product. For private interfaces, even if the implementation is always released, the IDL can still be maintained separately and the generated .c/.h files do not need to be released.

To take advantage of service object inheritance, the encode/decode library, version 5, and files generated from the QMI IDL compiler, version 5, are required along with two or more valid service objects from files generated by the QMI IDL compiler.

To inherit a parent service object, the function is:

```
int32_t qmi_idl_inherit_service_object
(
  qmi_idl_service_object_type child_service,
  qmi_idl_service_object_type parent_service
)
```

Where child_service is the service object used to register with the QCCI and QCSI.

NOTE: For service object inheritance to work, both the client and service sides must link the same service objects together in the same order.

One caveat to service object inheritance is that the parent service object must not have already inherited from another service object. This prevents the creation of a circular chain of inheritance that can potentially put the encode/decode library into an infinite loop.

For example, three service objects (A, B, and C) are to be linked together ($A \rightarrow B \rightarrow C$), where A is the child service object that is passed to the QCSI and QCCI.

To link the service objects together, the following functions are called:

```
qmi_idl_inherit_service_object(A,B);
qmi_idl_inherit_service_object(B,C);
```

However, if the functions are called in the opposite order:

```
qmi_idl_inherit_service_object(B,C);
qmi_idl_inherit_service_object(A,B);
```

The second call will fail because service object B has already inherited from service object C.

6.1 Sample code

Service object A and service object B have the same service ID. The following sample code shows how service B inherits from service A and registers with the QCSI.

```
qmi_idl_service_object_type  A_service_object =A_get_service_object();
qmi_idl_service_object_type  B_service_object =B_get_service_object();

qmi_idl_inherit_service_object(B_service_object, A_service_object);
//Child service object should be passed to qmi_csi_register.
qmi_csi_register(B_service_object, connect_cb,
     disconnect_cb, handle_req_cb, &service_cookie, os_params,
     &service_cookie.service_handle);
```

6.2 Inheritance feature for legacy services

The Inheritance feature is available for the QCSI only. There is no support for legacy services (e.g., QMI_NAS, QMI_WDS, QMI_WMS). As legacy services move to the QCSI, inheritance will be supported.

7 QMI VSS for Off-chip Communication

Currently, off-chip IPC communication is not supported. For example, in a Fusion 3 project with an MDM9x15 chipset and a Qualcomm Application-only Processor (APQ), IPC communication is supported inside the MDM9x15 chipset; however , it is not supported between the APQ and the MDM9x15 chipset. For the QMI communication between the APQ and the MDM9x15 chipset, QCSI services must register with the QMI Service Access Proxy (QSAP).

For customers who want to use the QCSI to create a VSS and the service is to be used by the QCCI running in an external application processor, the VSS must register with the QSAP. For details, refer to [Q6] .