ACKNOWLEDGEMENT

By utilizing this website and/or documentation, I hereby acknowledge as follows:

Effective October 1, 2012, QUALCOMM Incorporated completed a corporate reorganization in which the assets of certain of its businesses and groups, as well as the stock of certain of its direct and indirect subsidiaries, were contributed to Qualcomm Technologies, Inc. (QTI), a wholly-owned subsidiary of QUALCOMM Incorporated that was created for purposes of the reorganization.

Qualcomm Technology Licensing (QTL), the Company's patent licensing business, continues to be operated by QUALCOMM Incorporated, which continues to own the vast majority of the Company's patent portfolio. Substantially all of the Company's products and services businesses, including QCT, as well as substantially all of the Company's engineering, research and development functions, are now operated by QTI and its direct and indirect subsidiaries[1]. Neither QTI nor any of its subsidiaries has any right, power or authority to grant any licenses or other rights under or to any patents owned by QUALCOMM Incorporated.

No use of this website and/or documentation, including but not limited to the downloading of any software, programs, manuals or other materials of any kind or nature whatsoever, and no purchase or use of any products or services, grants any licenses or other rights, of any kind or nature whatsoever, under or to any patents owned by QUALCOMM Incorporated or any of its subsidiaries. A separate patent license or other similar patent-related agreement from QUALCOMM Incorporated is needed to make, have made, use, sell, import and dispose of any products or services that would infringe any patent owned by QUALCOMM Incorporated in the absence of the grant by QUALCOMM Incorporated of a patent license or other applicable rights under such patent.

Any copyright notice referencing QUALCOMM Incorporated, Qualcomm Incorporated, QUALCOMM Inc., Qualcomm Inc., Qualcomm or similar designation, and which is associated with any of the products or services businesses or the engineering, research or development groups which are now operated by QTI and its direct and indirect subsidiaries, should properly reference, and shall be read to reference, QTI.

[1] The products and services businesses, and the engineering, research and development groups, which are now operated by QTI and its subsidiaries include, but are not limited to, QCT, Qualcomm Mobile & Computing (QMC), Qualcomm Atheros (QCA), Qualcomm Internet Services (QIS), Qualcomm Government Technologies (QGOV), Corporate Research & Development, Qualcomm Corporate Engineering Services (QCES), Office of the Chief Technology Officer (OCTO), Office of the Chief Scientist (OCS), Corporate Technical Advisory Group, Global Market Development (GMD), Global Business Operations (GBO), Qualcomm Ventures, Qualcomm Life (QLife), Quest, Qualcomm Labs (QLabs), Snaptracs/QCS, Firethorn, Qualcomm MEMS Technologies (QMT), Pixtronix, Qualcomm Innovation Center (QuIC), Qualcomm iSkoot, Qualcomm Poole and Xiam.

QMI Vendor-Specific Services

User Guide

80-VM274-1 D

October 4, 2010

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

Qualcomm Confidential and Proprietary

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA 92121-1714
U.S.A.

Contents

Figures

Tables

Revision history

Revision	Date	Description
A	Aug 2008	Initial release
B	May 2009	Made editing changes to conform to Qualcomm standards
C	Dec 2009	Added Section 3.4.4 as a reference for vendors
D	Oct 2010	Modified vendor-specific service ID range

1 Introduction

1.1 Purpose

The QMI framework supports QMI vendor-specific services that can be added by vendors. With the QMI APIs, vendors can use the QMI service library to access the underlying QMI framework. These APIs are subject to change in future target releases.

1.2 Scope

This document provides the following details about QMI vendor-specific service support:

- Theory of operation
- User guide for vendors
- Semantics of exposed QMI APIs

1.3 Conventions

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

If you are viewing this document using a color monitor, or if you print this document to a color printer, **red boldface** indicates sample code snippets for reference.

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

1.4 References

Reference documents, which may include QUALCOMM®, standards, and resource 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		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1
Q2	*Qualcomm MSM™ Interface (QMI) Architecture*	80-VB816-1
Q3	*QMI Global Constant Definitions*	80-VB816-2

1.5 Technical assistance

For assistance or clarification on information in this guide, submit a case to Qualcomm CDMA Technologies at https://support.cdmatech.com/.

If you do not have access the CDMA Tech Support Service web site, register for access or send email to support.cdmatech@qualcomm.com.

1.6 Acronyms

For definitions of terms and abbreviations, see [Q1].

2 Theory of Operation

QMI vendor-specific service support enables vendors to add their own QMI services to run over Qualcomm's QMI framework. Utility functions are provided in QMI to enable vendors to register and use their vendor-specific services with the QMI framework.

Vendors can add their own QMI services and messages to their services. They can receive requests for their services and send responses/indications to the client using the QMI framework and the QMI common service library APIs.

Figure 2-1 is an example of vendor-specific service support.

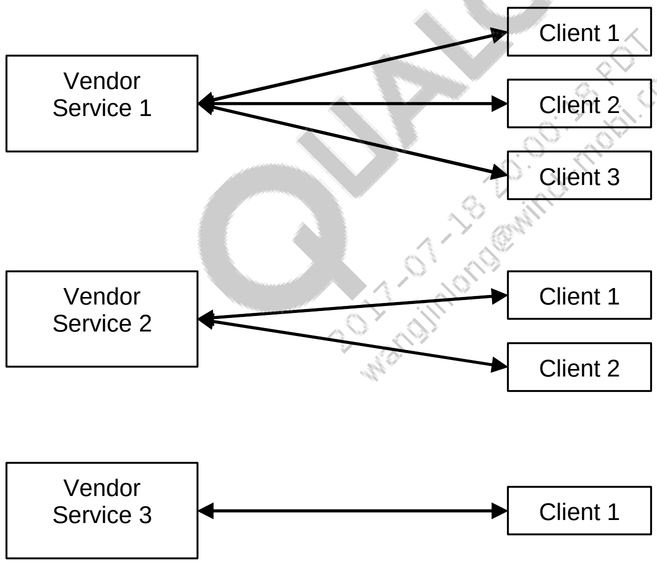

Figure 2-1 Vendor-specific service support

3 Implementing Vendor-Specific Services

A maximum of 27 vendor-specific services can be added by vendors. This chapter describes how to implement QMI vendor-specific services.

3.1 Service ID assignment

Every vendor-specific service must have a service ID. The valid vendor-specific service ID range is 227 to 253. Service ID assignment should be done inside the enum qmux_service_e_type in ds_qmi_svc_ext.h. Vendor-specific service IDs must be placed between QMUX_SERVICE_VENDOR_MIN and QMUX_SERVICE_VENDOR_MAX in the enum.

Vendor-specific services must be assigned service IDs consecutively and must begin with QMUX_SERVICE_VENDOR_MIN. QMUX_SERVICE_VENDOR_MAX should always be QMUX_SERVICE_VENDOR_MIN + the number of vendor-specific services. For example, if there are no vendor-specific services, then QMUX_SERVICE_VENDOR_MAX has a value of QMUX_SERVICE_VENDOR_MIN.

If you are viewing this document using a color monitor, or if you print this document to a color printer, **red boldface** indicates sample code snippets for reference.

Sample code in ds_qmi_svc_ext.h

```
typedef enum
{
QMUX_SERVICE_MIN      = 0,
/* Qualcomm Internal Service IDs */

QMUX_SERVICE_VENDOR_MIN  = 227,
VENDOR_SPECIFIC_SERVICE1     = QMUX_SERVICE_VENDOR_MIN,
VENDOR_SPECIFIC_SERVICE2     = QMUX_SERVICE_VENDOR_MIN +1,
 QMUX_SERVICE_VENDOR_MAX   //(= QMUX_SERVICE_VENDOR_MIN +2)
} qmux_service_e_type;
```

3.2 Service list

Multiple vendor-specific services are maintained using a service list data structure that is a two-dimensional array of qmux_svc_info_type. This service list can be used to map a specific set of services to each QMI instance. Table 3-1 is an example of QMI instances that are mapped to different sets of vendor-specific services as desired by the vendor.

Table 3-1 Example of QMI instances that support vendor-specific services

QMI instance	Vendor-specific services					
QMI INSTANCE 1	VS1	VS2	VS3	VS4	VS5	VS6
QMI INSTANCE 2	VS1	VS2	VS3	VS6	…	…
QMI INSTANCE 3	VS1	VS2	VS3	VS4	VS6	…

Memory must be allocated for the service list in ds_qmi_svc_ext.c. There should be enough memory allocated to facilitate all the services supported by the particular instance to which it is assigned. The maximum number of QMI instances is defined by the constant QMI_INSTANCE_MAX in ds_qmi_svc_ext.h. However, if the vendor-specific services are needed only on one QMI instance, then memory can be allocated only for one instance.

```
qmux_svc_info_type
qmi_vs_svc_list [number of qmi instances][number of VS services];
```

After the memory is allocated, the service list must be registered with QMUX, as described in Section 3.2.1.

3.2.1 Service list registration with QMUX

The qmux_reg_vs_service_list () function (declaration can be found in ds_qmux_ext.h) must be called to register the service list with QMUX. This function must be called separately for each QMI instance. Hence, the best place to call this function would be from within the qmi_svc_ext_init() function in ds_qmi_svc_ext.c, which is typically called separately for each QMI instance.

Vendors can selectively choose which QMI instance includes or excludes a particular service by selectively calling the service initialization function only on the desired QMI instance. This model gives the vendor the flexibility to choose not to register a service for a specific instance.

The following example shows how to register vendor-specific services for the first QMI instance. The check for the first instance (i==0) can be removed when support for multiple instances is added to the QMI framework.

Template for service-list registration

```
void qmi_svc_ext_init ( qmi_instance_e_type i )
{
  /* Built-in QMI Service Initializations */
…

  /* Vendor Specific Service List Registration with QMUX */
boolean reg_list_success;
if (i == 0) //check for 1st QMI instance
{
  reg_list_success = qmux_reg_vs_service_list(
                             i,
                             &(qmi_vs_svc_list[i][0]),
                             number_of_vendor_specific_services);
/* NOTE: Only after this step the vendor service initialization functions
can be called for individual services to register with QMUX. */

    /* Vendor Specific Services Initializations */
    vendor_service1_init(i);
    vendor_service2_init(i);
  …
}
}
```

3.3 Service initialization

Every service must have an initialization function (typically defined in the vendor-specific service source file), which should be called after the service list registration with QMUX is done. The following template shows the steps that must be performed in this function, followed by vendor-specific steps. A more detailed explanation of the APIs and structures used within it are described in Sections 3.3.1 through 3.3.3.

Initialization function template

```
void vendor_service_init( qmi_instance_e_type  qmi_inst )
{
  qmux_svc_info_type * qmux_svc_handle;
  qmi_vendor_service_state_type * vs1_sp;
  qmi_vendor_client_state_type * cl_sp;

  /* 1. Obtain Service Handle */
  qmux_svc_handle = qmux_reg_vs_service(qmi_inst,
                                       VENDOR_SERVICE_ID,
                                       &qmi_vs_cfg[qmi_inst]);
```

```
   /* 2. QMI Common Service State Initialization */
  vs1_sp = &qmi_vendor_service_state[qmi_inst];
   qmi_svc_common_svc_init(&vs1_sp->common, qmux_svc_handle);

   /* 3. Vendor specific service state initialization */
     …

   cl_sp = vs1_sp->client;
   for (client=0; client < VS1I_MAX_CLIDS; client++, cl_sp++)
   {
     /* 4. QMI Common Client State Initialization */
     qmi_svc_common_cl_init(&vs1_sp->common, &cl_sp->common);

     /* 5. Vendor specific client state initialization */
     …
   }
}
```

3.3.1 Service registration with QMUX

All individual services must be registered with QMUX using the qmux_reg_vs_service() function (the declaration can be found in ds_qmux_ext.h), which can be called from the initialization functions of the individual services, as shown in Section 3.3. This function must be called separately for each QMI instance and it will return a unique service handle (qmux_svc_handle in the initialization template**),** which can be used for the common service initialization.

The QMI vendor-service configuration, qmi_vs_cfg (see Section 3.3.2), must also be passed as one parameter in this function. Hence, the callback and the state pointers must be populated using the qmux_svc_config_type structure before this step.

3.3.2 Service configuration parameters (qmi_vs_cfg)

Vendors must allocate memory and set the members of the service configuration structure (qmux_svc_config_type) as needed in the vendor-specific service source file. Vendors can choose not to register their own implementation of the configuration members by defining them as NULL or zero appropriately. This will result in usage of the QMI built-in capability. The service configuration consists of the fields described in this section.

- Base/addendum versions – These can be optionally used by vendors to keep track of versions.
- Registerable callback function array (qmux_svc_cbs_type) – Vendors can register their own callback functions in this structure (qmux_svc_cbs_type). Choose to use the QMI built-in functionality by setting any of these pointers to NULL. Some of the callbacks shown below (in red) must be provided by the vendor, as they depend on vendor-specific implementation.

If you are viewing this document using a color monitor, or if you print this document to a color printer, **red boldface** indicates sample code snippets for reference.

```
typedef struct _qmux_svc_callbacks
{
  /* To allocate client IDs */
 byte (* alloc_clid) ( void * sp );

 /* To check if clid is valid */
  boolean (* is_valid_clid) ( void * sp, byte clid );

  /* To deallocate client IDs. */
 boolean (* dealloc_clid) ( void * sp, byte clid );

 /* To free all clients when QMI link is closed */
 void (* qmux_closed) ( void * sp );

  /* To get the QMUX SDU and trigger the appropriate command handler */
 void(* recv) (void * sp, byte clid, dsm_item_type * sdu_in);

/* To get the next client state pointer based on the client state pointer
passed as the  input to this function. Vendor needs to implement this as
the client state varies based on vendor specific definition.  */
 void * (* get_next_client_sp) ( void * sp, void * cl_sp);

/* To clean up the vendor specific client state. Vendor needs to implement
this function. This function can call qmi_svc_common_reset_client() api to
clean the common client state. */
  void (* reset_client) ( void *);

} qmux_svc_cbs_type;
```

QMI has built-in client ID management for clients. Vendors can choose to use it by setting the alloc_clid, is_valid_clid, dealloc_clid, qmux_closed, recv function pointers to NULL in the service configuration. However, vendors must register their own get_next_client_sp and reset_client callback functions in the service configuration. QMI needs the client state pointers to perform client ID management for each client. The client state pointers must be returned via get_next_client_sp callback, as the vendor client state can vary in size based on the vendor-specific parameters in it. The reset_client callback must clean up the client state and will trigger when the client ID is deallocated. The reset_client callback function can, additionally, call qmi_svc_common_reset_client(void * cl_sp_in) API to clean up the common client based on the common client-state pointer, which is passed in as a parameter.

- Command handler array – Any vendor-specific service must offer its services via commands that the client can use to request information that it needs. Every command must have a *unique command value* (e.g., CMD1 in the following example). The QMI framework gives vendors the ability to declare an array of command handlers (of type qmi_svc_cmd_hdlr_type defined in ds_qmi_svc_ext.h), which can be included in the vendor-specific service configuration.

Template

```
static qmi_svc_cmd_hdlr_type vs_cmd_handler_array[# of cmds] =
{
  {CMD1, “cmd description”,
                        (qmi_svc_hdlr_ftype) cmd_funtion_ptr)},
    …
};
```

- Number of commands – Number of commands supported in the service
- Service-provided state pointer – The following example declares the service configuration structure for one instance:

```
qmux_svc_config_type qmi_vs_cfg[1] =
{
  { { base major version, base minor version },
     { addendum major version, addendum minor version },
    { NULL, // alloc: use QMI built-in capability
      NULL, // is_valid_clid: use QMI built-in capability
      NULL, // dealloc: use QMI built-in capability
      NULL, // qmux_closed: use QMI built-in capability
      NULL, // recv: use QMI built-in capability
      qmi_vendor_service_get_next_client_sp,
      qmi_vendor_service_reset_client
    },
    vs_cmd_handler_array,
    # of vendor service commands (in the array),
    &qmi_vendor_service_state[0]
 };
```

3.3.3 QMI common service and client state initializations

Every vendor-specific service must maintain a state (qmi_vendor_service_state[], in the vendor-specific service source file) to store service-related information. A vendor-specific service can have multiple clients to which it can offer services, and for which it must maintain a client state to store information that applies to specific clients. The client and service state structures must follow the format described in this section to work with the QMI model.

- Vendor-specific service client state definition (cl_sp)

```
typedef struct qmi_vendor_service_client_state_s
{
  byte common[QMI_COMMON_CLIENT_STATE_SIZE]; //must be first
 /* Add vendor-defined client specific fields here. Eg: Number of times a
request was serviced for the client */
} qmi_vendor_service_client_state_type;
```

- Vendor-specific service state definition (vs1_sp)

```
typedef struct qmi_vendor_service_state_s
{
  byte common[QMI_COMMON_SVC_STATE_SIZE]; // must be first
  qmi_vendor_service_client_state_type client[num of clients];
  /* Add vendor specific fields here. Eg: Total number of request messages
serviced so far */
} qmi_vendor_service_state_type;
```

Vendors are required to declare and allocate memory for the service state for any number of QMI instances as desired; for example, of memory allocation for one instance:

```
static qmi_vendor_service_state_type  qmi_vendor_service_state[1] = {0,};
```

The common service and client states in the above structures must be initialized using qmi_svc_common_svc_init() and qmi_svc_common_cl_init() APIs, as shown in the initialization function template in Section 3.3. The QMI framework supports a maximum of 255 clients per service and the number of elements in the client[] array in the vendor-specific service state cannot exceed 255. The qmi_svc_common_cl_init() must be called separately for each client. Vendor-specific service-state and vendor-specific client-state initializations can be performed after the common-state initializations.

3.4 QMI command handling

Vendor-specific services must offer their services via commands that the client can use to request the information it needs. Each command can be associated with the corresponding vendor-defined command handler function. An array of command handlers (vs_cmd_handler_array) of type qmi_svc_hdlr_ftype (defined in ds_qmi_svc_ext.h) must be defined as part of the service configuration parameters, described in Section 3.3.2, in the vendor-specific service source file.

```
typedef dsm_item_type* (* qmi_svc_hdlr_ftype)
               (
                 void * sp,      //Service provided state pointer
                 void * cmd_buf_p, // Command buffer
                 void * cl_sp,    // Client state pointer
                 dsm_item_type ** sdu //Packet containing request to be
processed
               );
```

The sp and cl_sp parameters should be appropriately cast to the vendor-specific service-state and client-state structures, respectively, in the command handlers. The cmd_buf_p parameter holds the command information used by the QMUX layer. The sdu parameter is the dsm_item_type packet that can hold any input parameters specific to a command. The data from this packet must be extracted using the DSM APIs.

Vendor-specific services have the option to make the command handlers blocking or nonblocking.

3.4.1 Blocking request

This option should be used when the service can immediately process the request. The response must be pushed into a dsm_item_type packet in the command handler function, and this packet should be returned by the command handler.

See Section 3.4.4 for an example of how to build a response packet.

3.4.2 Nonblocking request

The vendor-specific service must store the command buffer pointer (cmd_buf_p) and return 1 as the return value for the command handler. This is a way to inform the QMI framework that the response for the request is pending and will be sent at a later time. Once the response is ready, the vendor-specific service can retrieve the previously stored command buffer and use the qmi_svc_send_response() API to send the response to the client.

```
boolean qmi_svc_send_response
(
   void * sp, // Service-provided state pointer
   void * cmd_buf_p_in, // Retrieved command buffer pointer
   dsm_item_type *  msg_ptr   // Response packet
);
```

See Section 3.4.4 for an example of how to build a response packet.

The qmi_svc_get_client_sp_by_cmd_buf() API returns the client-state pointer from the command buffer, which is de-queued. This can be used to send the response to specific clients, if required, by checking if this client state matches the desired client.

```
void * qmi_svc_get_client_sp_by_cmd_buf
(
  void * cmd_buf_p_in //Retrieved command buffer pointer
);
```

3.4.3 Asynchronous indications

In addition to synchronous queries and responses, indications can be sent asynchronously by the vendor-specific service using the qmi_svc_send_ind() API.

```
void qmi_svc_send_ind
(
  void * sp,  // Service-provided state pointer
  uint8 clid,// Client ID of the client to send the ind
  uint16 cmd_type, //Command value/ID associated with this  ind
  dsm_item_type * ind //Packet containing indication info to be sent
);
```

This API requires client ID and cmd_type input parameters in addition to the service-state pointer and data packet to be sent in the indication. The cmd_type parameter is the unique command value described in Section 3.3.2. For broadcast indications sent out to all clients, the client ID must be set to 255. For unicast indications, the client ID of the client must be used. If vendors choose QMI to do the client ID management, they can use the qmi_svc_get_clid_from_client_sp() API to get the client ID that is associated with the client.

```
uint8 qmi_svc_get_clid_from_client_sp
(
  void * cl_sp  // Client state pointer whose client ID is returned via
this api
);
```

See Section 3.4.4 for an example of how to build a packet to be sent in the indication.

3.4.4 Building a response/indication packet

This section provides an example of how to build a packet to be sent in the response or indication messages that are described in Sections 3.4.1, 3.4.2, and 3.4.3. In this example, the packet being built contains a result TLV (only relevant for response and not indication) and another VS TLV that holds a 4-byte integer (int32). The response is built by pushing both TLVs from back to front. See [Q3] for example values of constants for QMI_ERR_* and QMI_RESULT_*.

The following psuedocode provides an example of how to build a response/indication packet:

```
static dsm_item_type * qmi_vs_build_packet()
{
   ...
   dsm_item_type * response = NULL; //assume memory allocation done
   uint16 errval = QMI_ERR_NONE;
   uint16 result = QMI_RESULT_SUCCESS;

   /* STEP 1: PUSH VS TLV INTO THE RESPONSE DSM ITEM */
   uint8 type = 1; //type of VS TLV
   uint16 len = sizeof(int32); //length of VS TLV
   int32 out = 3;  //value of VS TLV

   /* STEP 1a: PUSH VALUE OF VS TLV INTO THE RESPONSE */
   if (len != dsm_pushdown_packed(&response, &out,
                                  len, DSM_DS_SMALL_ITEM_POOL))
   {
        /* Cannot populate VALUE into response pointer */
        errval = QMI_ERR_*;
        dsm_free_packet(&response);
   }
   else
   {
        /* STEP 1b: PUSH LENGTH OF VS TLV INTO THE RESPONSE */
        if (sizeof(len) != dsm_pushdown_packed(&response, &len,
                                  sizeof(len), DSM_DS_SMALL_ITEM_POOL))
        {
             /* Cannot populate LENGTH into response pointer */
             errval = QMI_ERR_*;
             dsm_free_packet(&response);
        }
```

```
            else
            {
                /* STEP 1c: PUSH TYPE OF VS TLV INTO THE RESPONSE */
                 if (sizeof(type) != dsm_pushdown_packed(&response, &type,
                                        sizeof(type), DSM_DS_SMALL_ITEM_POOL))
                {
                    /* Cannot populate TYPE into response pointer */
                    errval = QMI_ERR_*;
                    dsm_free_packet(&response);
                }
            }
        }

        /* STEP 2: PUSH RESULT TLV INTO THE RESPONSE DSM ITEM */
        /* NOTE: Step 2 is not applicable for indication messages */
        type = 2; // type of Result TLV
        //length of Result TLV (sizes of errval + result)
        len = sizeof(uint16) + sizeof(uint16);
        if (errval == QMI_ERR_NONE)
        {
            result = QMI_RESULT_SUCCESS; /* value = Success */
        }
        else
        {
            result = QMI_RESULT_FAILURE; /* value = Failure */
        }

        /* STEP 2a: PUSH VALUE OF RESULT TLV INTO THE RESPONSE */
        dsm_pushdown_packed(&response, &errval,
                                sizeof(uint16), DSM_DS_SMALL_ITEM_POOL);
        dsm_pushdown_packed(&response, &result,
                                sizeof(uint16), DSM_DS_SMALL_ITEM_POOL);
        /* STEP 2b: PUSH LENGTH OF RESULT TLV INTO THE RESPONSE */
        dsm_pushdown_packed(&response, &len,
                                sizeof(len), DSM_DS_SMALL_ITEM_POOL);
        /* STEP 2c: PUSH TYPE OF RESULT TLV INTO THE RESPONSE */
        dsm_pushdown_packed(&response, &type,
                                sizeof(type), DSM_DS_SMALL_ITEM_POOL);

        return response;
    }
```

4 Limitations/Assumptions

The following limitations and assumptions apply:

- There is no deregister function supported in the current implementation that allows deregistration with QMUX.
- Vendor-specific services must be compiled with AMSS.
- Vendor services run either in the DS or DCC task context.