DSI_NetCtrl Library API

Interface Specification

80-NH200-1 B

June 19, 2013

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

Contents

List of Figures

List of Tables

Revision History

Revision	Date	Description
A	May 2013	Initial release.
B	Jun 2013	Autogenerated release with minor corrections.

1 Introduction

1.1 Purpose

This document describes the Data Services Interface Network Control (DSI_NetCtrl) library API. This API provides a common interface for clients to perform data services call control in the High Level Operating System (HLOS) environment.

1.2 Scope

This document describes the functionality and the interface of the DSI_NetCtrl library. This document is intended for developers involved in the development of software that requires WWAN data services functionality.

1.3 Conventions

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

Code variables appear in angle brackets. For example, `<number>`.

Parameter directions are indicated as follows:

- `[in]` indicates an input parameter.
- `[out]` indicates an output parameter.
- `[in,out]` indicates a parameter used for both input and output.

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 Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.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
Standards		
S1	*3GPP technical specification, Release 6*	3GPP TS 24.008 V6.9.0
S2	*Quality of Service (QoS) concept and architecture*	3GPP TS 23.107 V7.1.0
S3	*Administration of Parameter Value Assignments for cdma2000 Spread Spectrum Standards*	3GPP2 C.R101-G (TSB-58)

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].

2 Operational Description

2.1 Overview

The DSI_NetCtrl (Data Services Interface) library provides functionality for WWAN data call bring-up and tear-down, data call configuration, Quality of Service (QoS), enhanced Multimedia Broadcast/Multicast Services (eMBMS), and Domain Name Service (DNS). It is provided as a library that applications can link to to get data services functionality.

2.1.1 Functionality

The DSI_NetCtrl library provides the following functionality:

- Data call configuration – Allows applications to configure the data call parameters before call bring-up. Parameters such as Access Point Name (APN), IP family, technology preference, username/password, and profile ID can be configured for the data call.
- Data call bring-up and tear-down – DSI_NetCtrl supports bring-up and tear-down of data calls. There are event callbacks that notify success or failure of these operations.
- Data call status and auxiliary query methods – DSI_NetCtrl provides methods for querying the status of data calls initiated by the client. It also allows for several auxiliary functions, such as querying the data bearer technology, data rate, data packet statistics, etc.
- Dual-IP support – DSI_NetCtrl provides inherent support for dual-IP calls. Clients can use a single method to trigger dual-IP data calls and tear them down. Clients do not have to manage dual-IP data calls as two separate single-IP calls.
- Multi-PDN support – DSI_NetCtrl supports control plane bring-up of multiple Public Data Networks (PDN). Applications can bring as many PDNs as the modem can support. DSI_NetCtrl also provides functionality to perform DNS queries over these individually brought up PDNs. The application must take care of data path routing.
- QoS – DSI_NetCtrl provides methods for applications to request UE-initiated QoS. DSI_NetCtrl also informs applications of secondary flows installed as part of network-initiated QoS.
- Multiprocess support – DSI_NetCtrl can be linked to applications in different processes, and those individual processes can independently perform data call operations such as bring-up and tear-down. DSI_NetCtrl always relies on the modem state and can appropriately handle multi-PDN arbitration scenarios from different processes.

2.1.2 Limitations

Following are the known limitations for DSI_NetCtrl library.

- No support for data path functions – DSI_NetCtrl was designed to be a control pane management layer for data activities and therefore does not provide methods for data transfer explicitly. DSI_NetCtrl is supported for Linux-based platforms, so native data path methods such as Berkeley Software Distribution (BSD) or Unix sockets can be used together with control pane methods provided by DSI_NetCtrl.
- Multiprocess support – Currently, multiprocess support has been tested from up to three different processes only. There are modem Qualcomm Messaging Interface (QMI) resources that DSI_NetCtrl uses for each instantiation; hence support for more than three processes has not been tested so far. If support is required for more than three processes, enhancements would be needed in the DSI_NetCtrl library.

2.2 Theory of Operation

The use of the DSI_NetCtrl library is described in this section. The client links to the DSI_NetCtrl library, which is provided as a shared library on Qualcomm supported Unix-based operating systems. The library provides methods for client initialization, call configuration, call bring-up and tear-down, QoS, and DNS. Many of the calls are dependent on the modem and the network processing the request. Hence, the DSI_NetCtrl library provides a set of events that are asynchronously notified to the client.

2.2.1 Client Libray Initialization

The DSI_NetCtrl library must be initialized before being used in a process. The initialization method dsi_init() must be called before any other calls to the library. After dsi_init(), the client must perform a dsi_get_data_srvc_hndl() to get a handle to the DSI library. All further operations (e.g., dsi_start_data_call, dsi_stop_data_call, etc.) are called using this handle.

Figure 2-1 shows the call flow for library initialization.

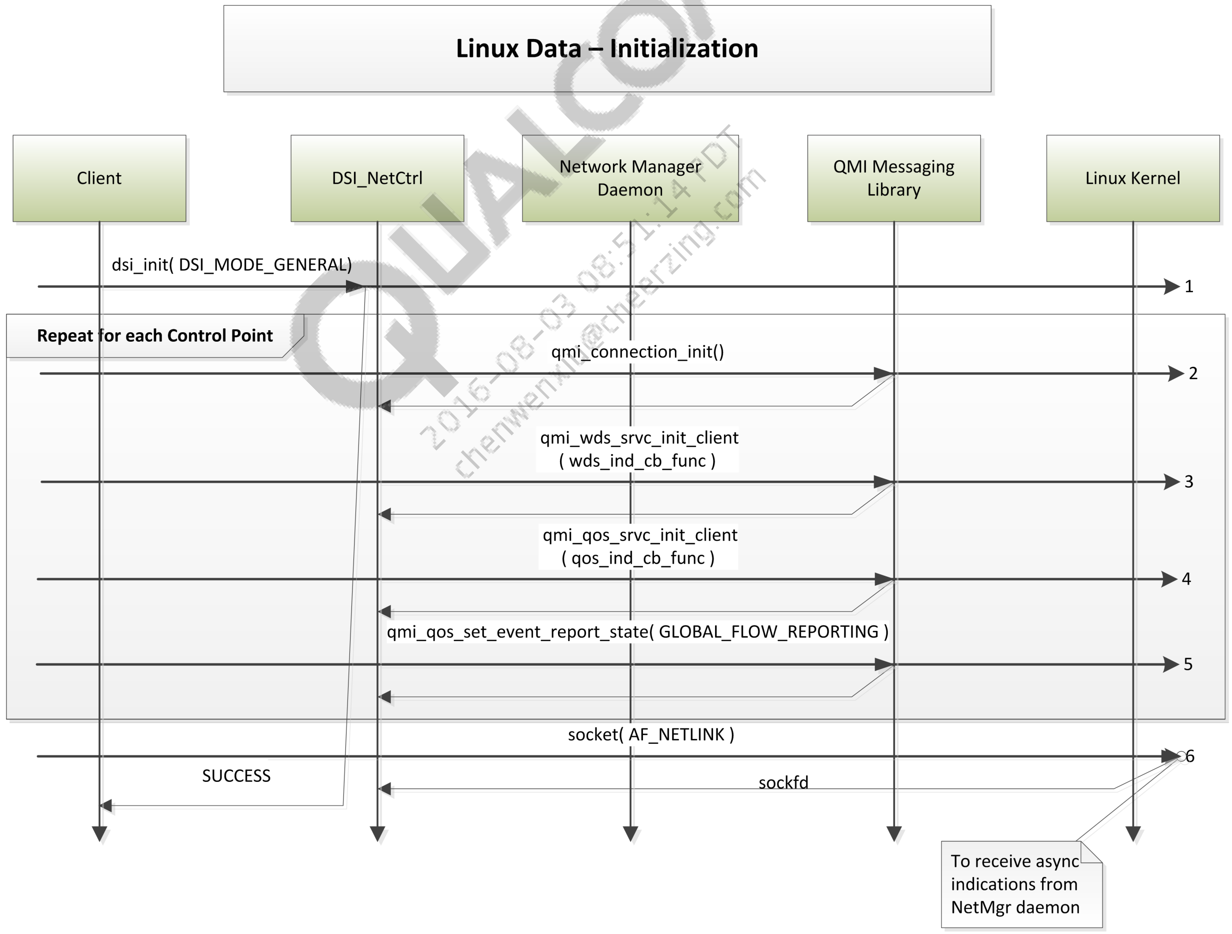
Linux Data – Initialization
Client
DSI_NetCtrl
Network Manager Daemon
QMI Messaging Library
Linux Kernel
dsi_init(DSI_MODE_GENERAL)
1
Repeat for each Control Point
qmi_connection_init()
2
qmi_wds_srvc_init_client (wds_ind_cb_func)
3
qmi_qos_srvc_init_client (qos_ind_cb_func)
4
qmi_qos_set_event_report_state(GLOBAL_FLOW_REPORTING)
5
socket(AF_NETLINK)
6
SUCCESS
sockfd
To receive async indications from NetMgr daemon

Figure 2-1 Client library initialization

2.2.2 Data Call Configuration and Setup

Figure 2-2 shows the client-initiated data call configuration and setup.

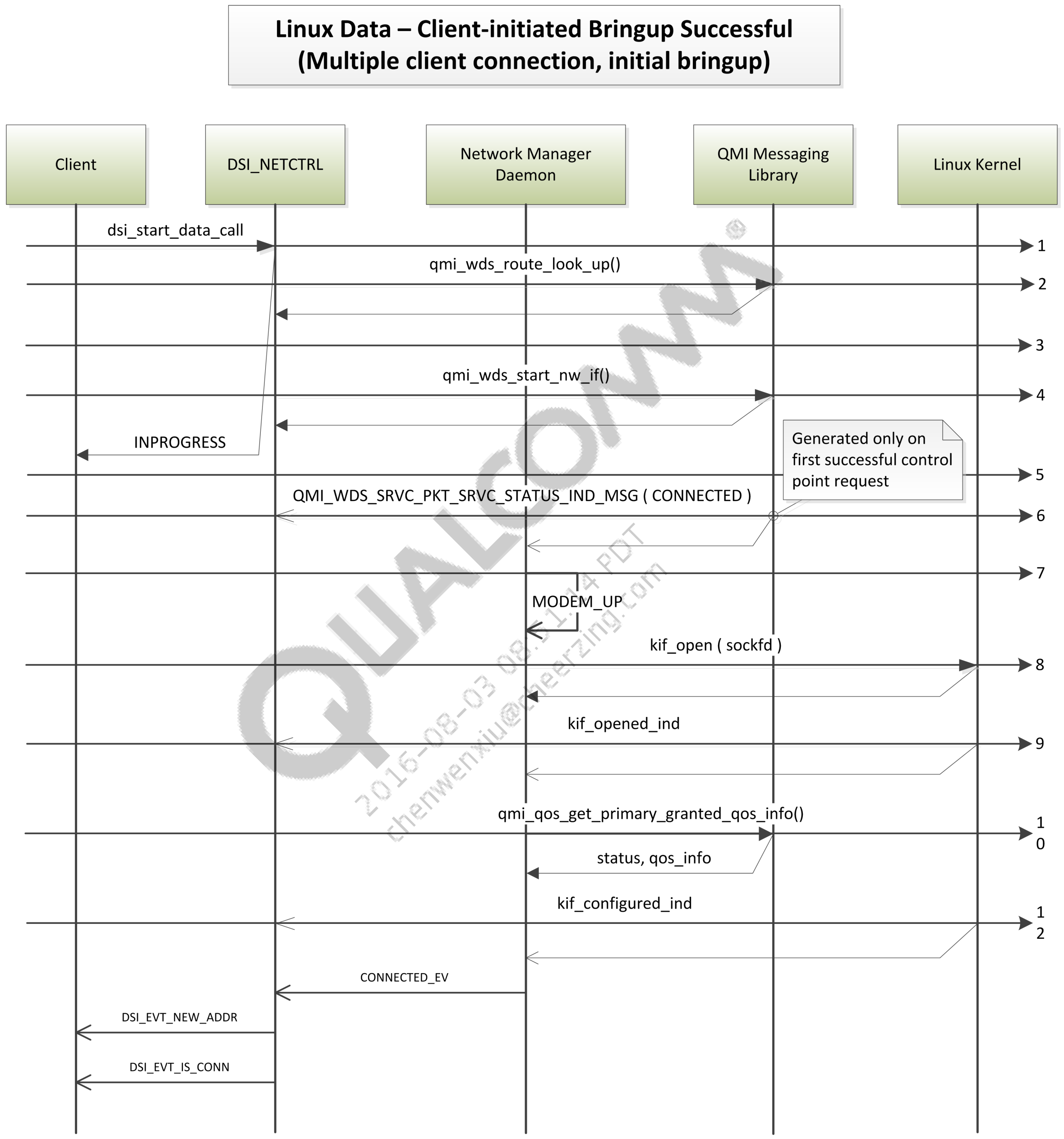

Figure 2-2 Client-initiated call setup

After dsi_init() has been performed, the client can configure the call bring-up parameters using dsi_set_data_call_params(). This function supports setting of the following parameters:

- UMTS/CDMA profile ID
- Access Point Name (APN)
- IP version (IPv4, IPv6, or IPv4v6)
- APN username and password
- Authentication preference
- Technology type/extended technology name

The configuration method can be called multiple times to set the required parameters. Once all the parameters are specified, dsi_start_data_call() must be called to start the data session. Depending on the configuration parameters, the DSI attempts to bring-up the call and indicate success, failure, or partial success (in the case of dual-IP) through event callbacks. Figure 2-2 shows call bring-up for a dual-IP scenario.

Call bring-up is based on best effort algorithm. The DSI first attempts to first match the profile to find the correct profile ID on the modem and execute a call bring-up on that profile. If the APN and IP family parameters are specified without a profile ID, the DSI attempts to find the profile using the APN and IP family parameters. If a profile ID matching the APN and IP family requirements is not found, the DSI internally creates a transient profile on the modem with the APN and IP family parameters set. It then uses this profile to bring up the data call.

For the case of single IP calls, after the call is established, the appropriate RmNet network interface is brought up by the lower Linux data layers (NETMGR in case of LE). The DSI receives the RmNet interface up indication through NETMGR, and conveys this to its clients (DSI_EVT_NEWADDR). Since a single interface was requested, the DSI also conveys DSI_EVT_IS_CONN to the application immediately after DSI_EVT_NEWADDR has been reported.

For dual-IP calls, after each IP family has been brought up, the DSI indicates corresponding DSI_EVT_NEWADDR events. However, DSI_EVT_IS_CONN is notified only after both the IP families are up.

2.2.3 Data Call Termination

Data call termination can be initiated by the client or by the network.

2.2.4 Client-initiated Data Call Termination

Figure 2-2 shows the client-initiated data call termination.

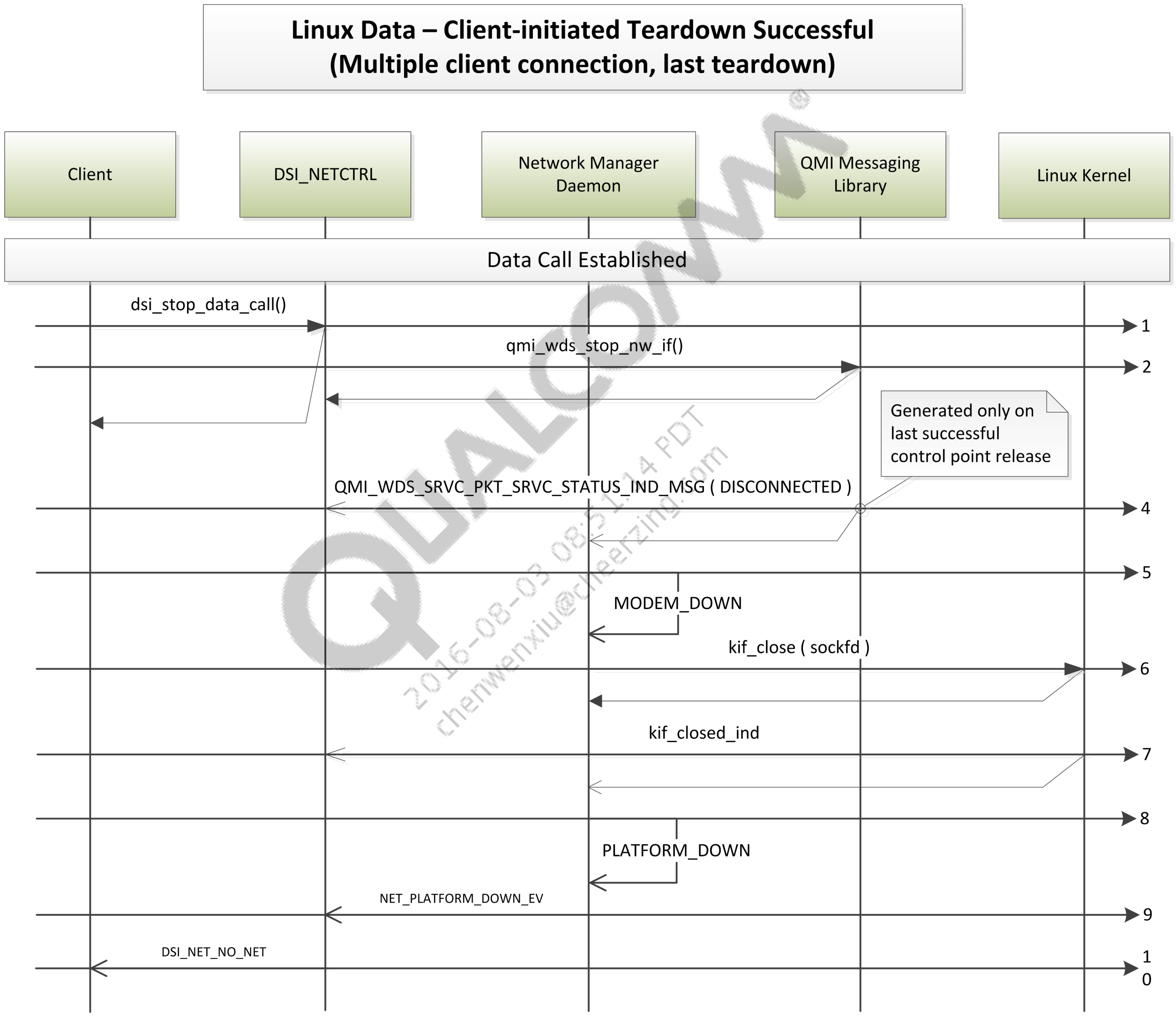

Figure 2-3 Client-initiated data call termination

DSI provides a method for clients to terminate the data call that it originated: dsi_stop_data_call(). The above call flow indicates how the data call is torn down in the case of dual-IP scenario.

For a single IP scenario, after the call is torn down, a DSI_EVT_DELADDR event is posted along with DSI_EVT_NO_NET. For a dual-IP scenario, after each family is torn down DSI_EVT_DELADDR is posted. After both families are torn down, DSI_EVT_NO_NET is posted.

2.2.5 Network-initiated Data Call Termination

Figure 2-4 shows the network-initiated data call termination.

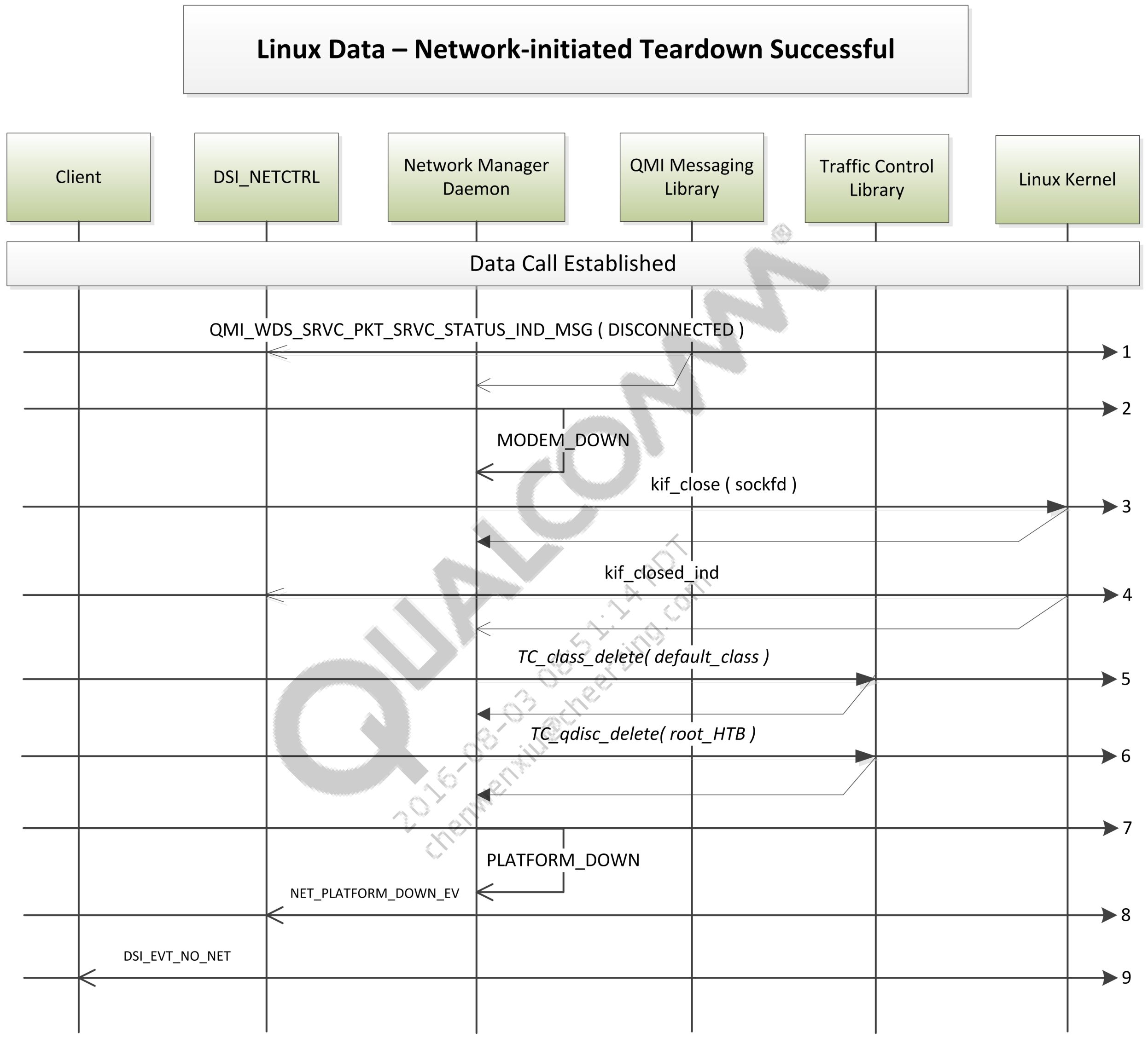

Figure 2-4 Network-initiated call termination

In some scenarios, the UE may move to out-of-service or may move to another technology that does not support IP continuity handover. In such a scenario the data call is torn down by the network. DSI receives this information through the QMI_WDS service, performs local cleanup, and relays this information to its clients. Figure 2-4 illustrates how this behavior is accomplished.

2.2.6 Quality of Service

DSI_NetCtrl supports Quality of Service (QoS) operations (both UE-initiated and network-initiated QoS are supported). The following is the theory of operation for QoS. For any operations with a secondary QoS flow, the primary connection must be brought up first.

Note: QoS may or may not be enabled on all product lines.

2.2.7 Request QoS Flow on a Network Connection

Figure 2-5 shows the client-initiated QOS request.

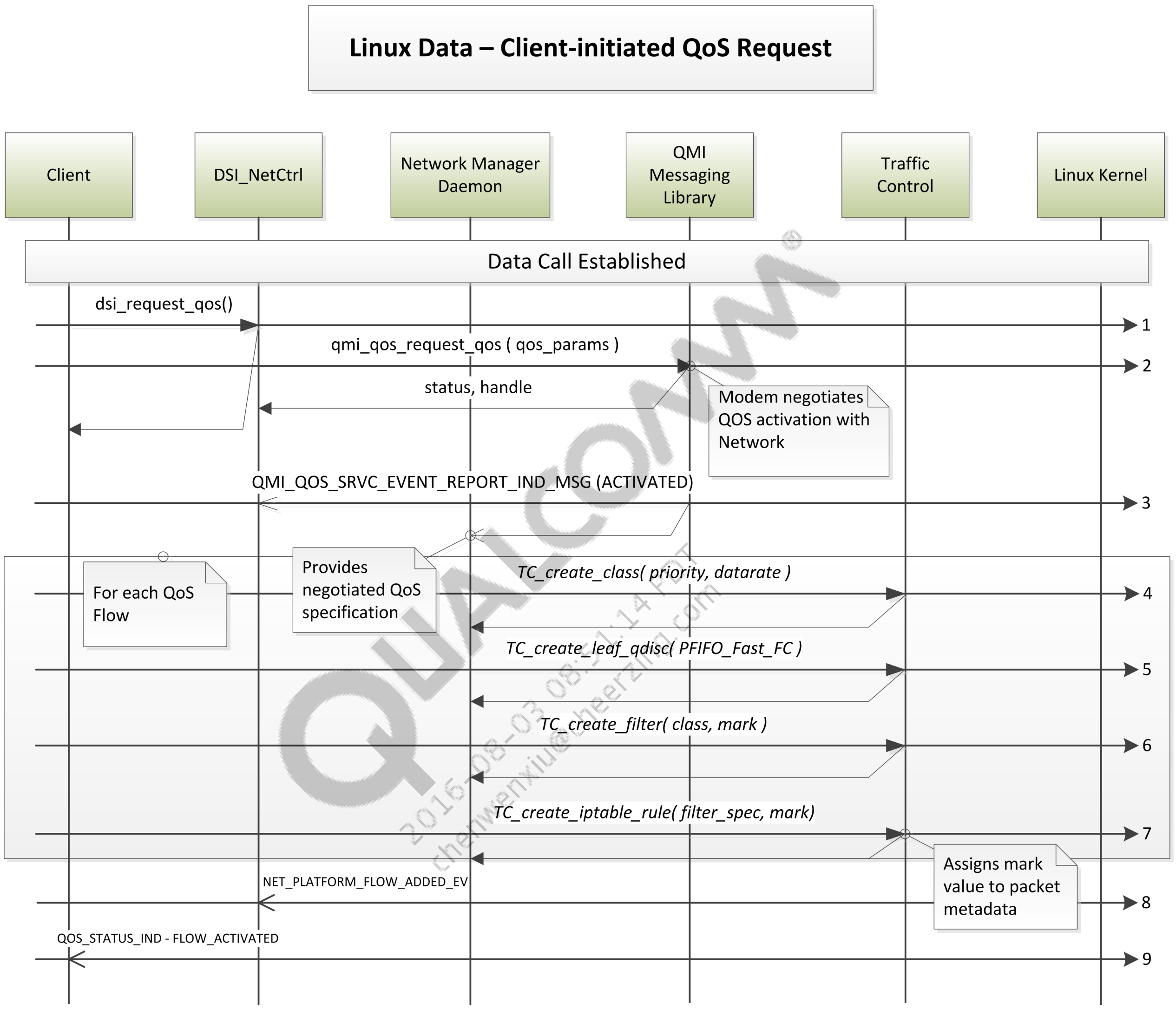
Linux Data – Client-initiated QoS Request
Client
DSI_NetCtrl
Network Manager Daemon
QMI Messaging Library
Traffic Control
Linux Kernel
Data Call Established
dsi_request_qos()
1
qmi_qos_request_qos (qos_params)
2
status, handle
Modem negotiates QOS activation with Network
QMI_QOS_SRVC_EVENT_REPORT_IND_MSG (ACTIVATED)
3
For each QoS Flow
Provides negotiated QoS specification
TC_create_class(priority, datarate)
4
TC_create_leaf_qdisc(PFIFO_Fast_FC)
5
TC_create_filter(class, mark)
6
TC_create_iptable_rule(filter_spec, mark)
7
Assigns mark value to packet metadata
NET_PLATFORM_FLOW_ADDED_EV
8
QOS_STATUS_IND - FLOW_ACTIVATED
9

Figure 2-5 Client-initiated QoS request

After bringing up the data call, the client can bring up a UE-initiated QoS flow. This operation is supported only if the network supports a secondary bearer. From the client perspective, the client calls dsi_request_qos() which takes the flow and filter specifications as input parameters. Once the QoS is granted, clients receive DSI_EVT_QOS_STATUS_IND to indicate a QoS-related activity has happened. DSI_EVT_QOS_STATUS_IND contains one of the following status parameters:

- DSI_QOS_ACTIVATED_EV
- DSI_QOS_SUSPENDED_EV
- DSI_QOS_GONE_EV
- DSI_QOS_MODIFY_ACCEPTED_EV
- DSI_QOS_MODIFY_REJECTED_EV

- DSI_QOSINFO_CODE_UPDATAED_EV
- DSI_QOS_FLOW_ENABLED_EV
- DSI_QOS_FLOW_DISABLED_EV

The callflow in Figure 2-5 illustrates how a client-initiated QoS operation works. After the QoS control pane is up, the netmgr framework internally installs the queue disciplines, so the QoS flows can be appropriately prioritized.

2.2.8 Client-initiated QoS Release

Figure 2-6 shows the client-initiated QoS release.

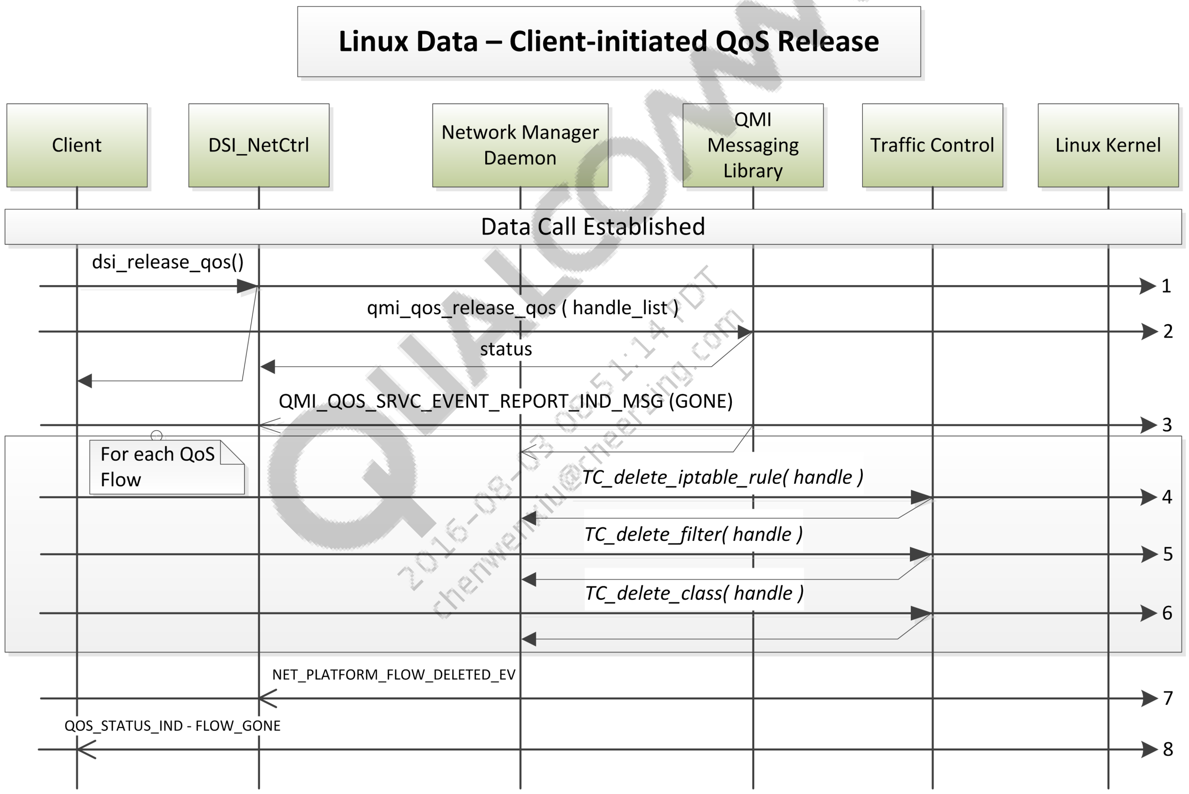
Linux Data – Client-initiated QoS Release
Client
DSI_NetCtrl
Network Manager Daemon
QMI Messaging Library
Traffic Control
Linux Kernel
Data Call Established
dsi_release_qos()
1
qmi_qos_release_qos (handle_list)
2
status
QMI_QOS_SRVC_EVENT_REPORT_IND_MSG (GONE)
3
For each QoS Flow
TC_delete_iptable_rule(handle)
4
TC_delete_filter(handle)
5
TC_delete_class(handle)
6
NET_PLATFORM_FLOW_DELETED_EV
7
QOS_STATUS_IND - FLOW_GONE
8

Figure 2-6 Client-initiated QoS release

The client can release the QoS connection by calling dsi_release_qos(). After the QoS flow is released, all associated filters installed with the kernel are removed by the netmgr framework. Once QoS is successfully released, DSI_EVT_QOS_STATUS_IND is sent with QOS_STATUS_DEACTIVATED to clients.

2.2.9 Modify QoS Flow on a Network Connection

Figure 2-7 illustrates the acceptance of a client-initiated modification to QOS.

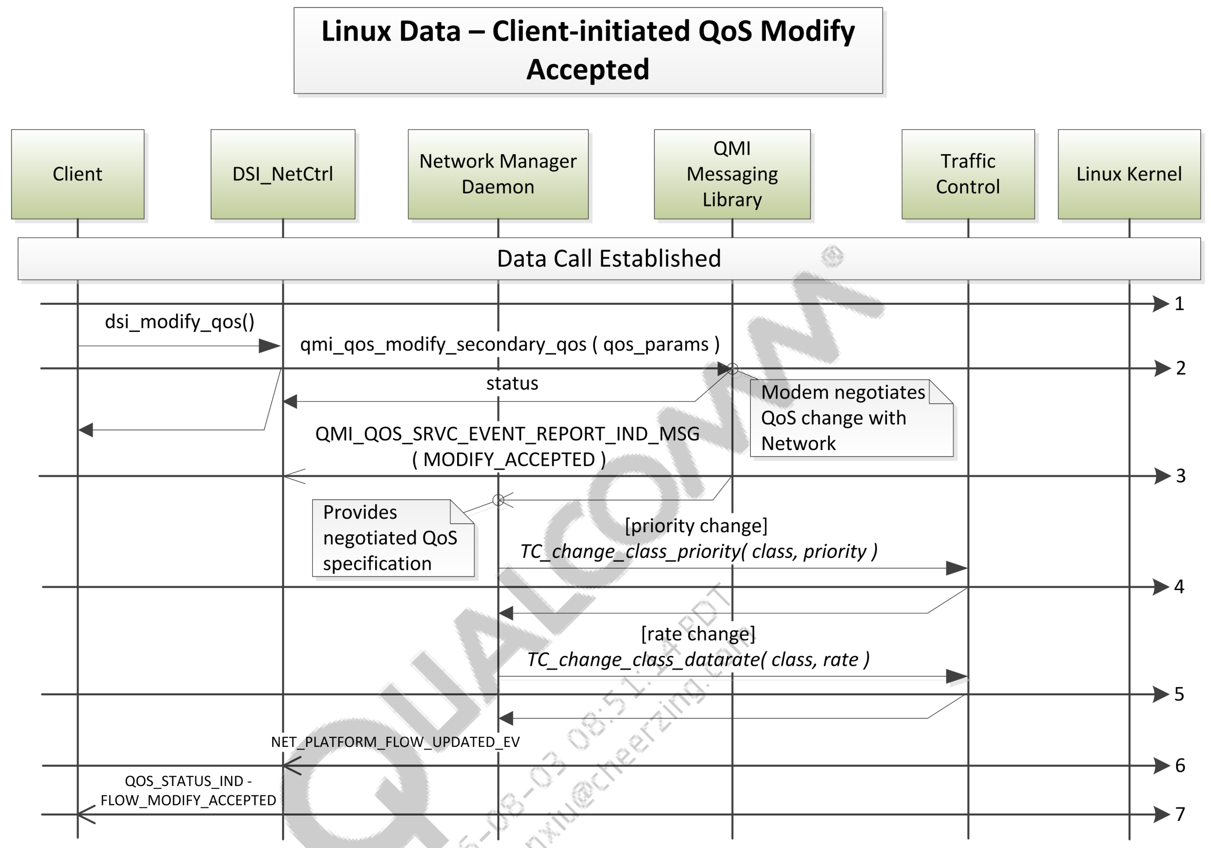
Linux Data – Client-initiated QoS Modify Accepted
Client
DSI_NetCtrl
Network Manager Daemon
QMI Messaging Library
Traffic Control
Linux Kernel
Data Call Established
1
dsi_modify_qos()
qmi_qos_modify_secondary_qos (qos_params)
2
status
Modem negotiates QoS change with Network
QMI_QOS_SRVC_EVENT_REPORT_IND_MSG (MODIFY_ACCEPTED)
3
Provides negotiated QoS specification
[priority change]
TC_change_class_priority(class, priority)
4
[rate change]
TC_change_class_datarate(class, rate)
5
NET_PLATFORM_FLOW_UPDATED_EV
6
QOS_STATUS_IND - FLOW_MODIFY_ACCEPTED
7

Figure 2-7 Client-initiated QoS modify accepted

The client can request modification of a QoS flow specification by using the dsi_modify_qos() method. The handle for which the modification is being requested must be obtained before requesting dsi_request_qos(). The new specification must be provided as part of the parameters to this method.

2.2.10 Suspend/Resume QoS

Figure 2-8 shows the client-initiated suspension/resumption of a QoS flow.

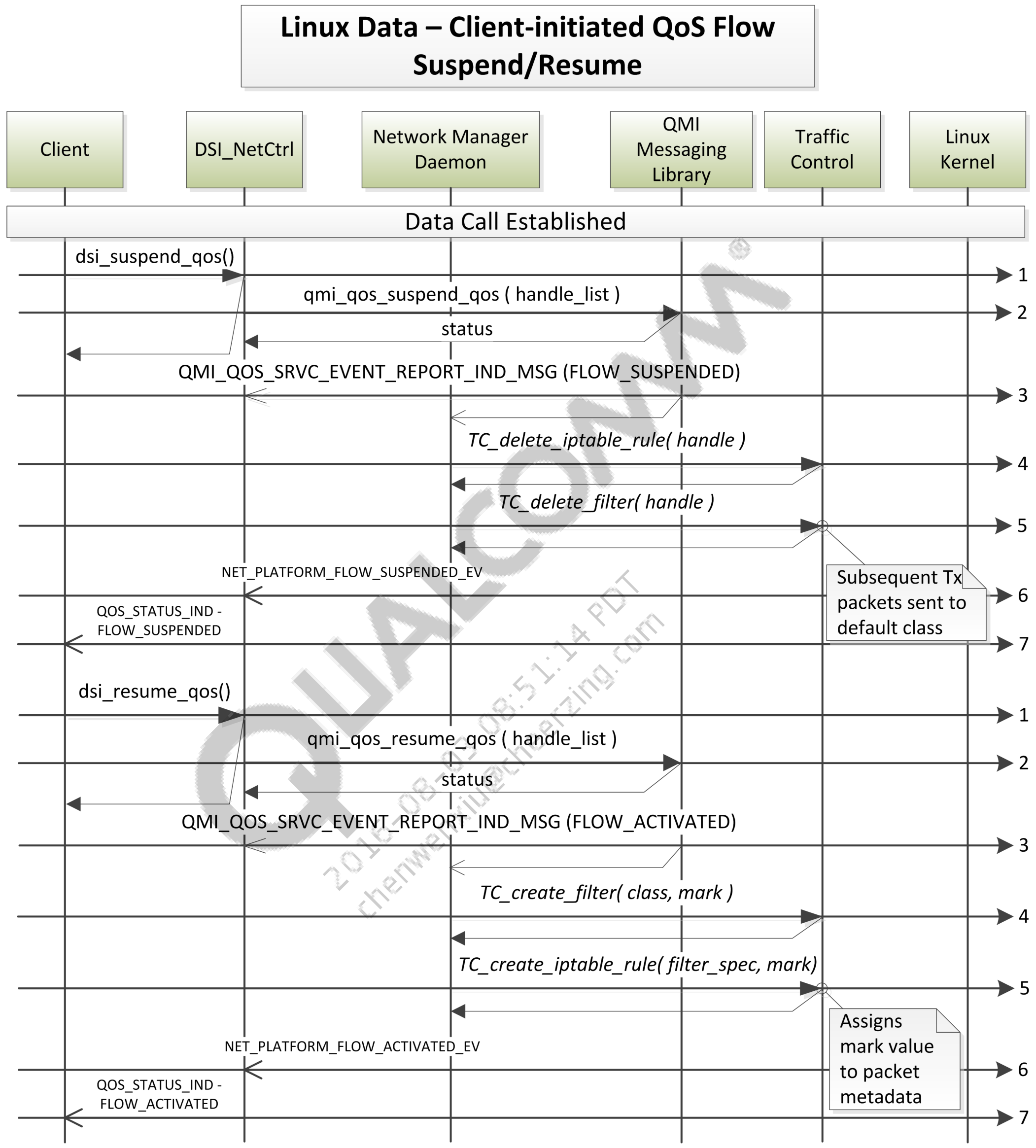
Linux Data – Client-initiated QoS Flow Suspend/Resume
Client
DSI_NetCtrl
Network Manager Daemon
QMI Messaging Library
Traffic Control
Linux Kernel
Data Call Established
dsi_suspend_qos()
qmi_qos_suspend_qos (handle_list)
status
QMI_QOS_SRVC_EVENT_REPORT_IND_MSG (FLOW_SUSPENDED)
TC_delete_iptable_rule(handle)
TC_delete_filter(handle)
NET_PLATFORM_FLOW_SUSPENDED_EV
QOS_STATUS_IND - FLOW_SUSPENDED
Subsequent Tx packets sent to default class
dsi_resume_qos()
qmi_qos_resume_qos (handle_list)
status
QMI_QOS_SRVC_EVENT_REPORT_IND_MSG (FLOW_ACTIVATED)
TC_create_filter(class, mark)
TC_create_iptable_rule(filter_spec, mark)
NET_PLATFORM_FLOW_ACTIVATED_EV
QOS_STATUS_IND - FLOW_ACTIVATED
Assigns mark value to packet metadata
1
2
3
4
5
6
7

Figure 2-8 Client-initiated QoS suspend/resume

The client can suspend a QoS operation temporarily with dsi_suspend_qos() and resume the QoS operation with dsi_resume_qos(). The call flow in Figure 2-8 illustrates how suspend and resume indications work. From an end-client perspective, QOS_STATUS indications are generated in both cases after successful execution of suspend or resume.

2.2.11 Query-granted QoS Information on a Network Connection

Figure 2-9 shows the call flow for a query for QoS flow information.

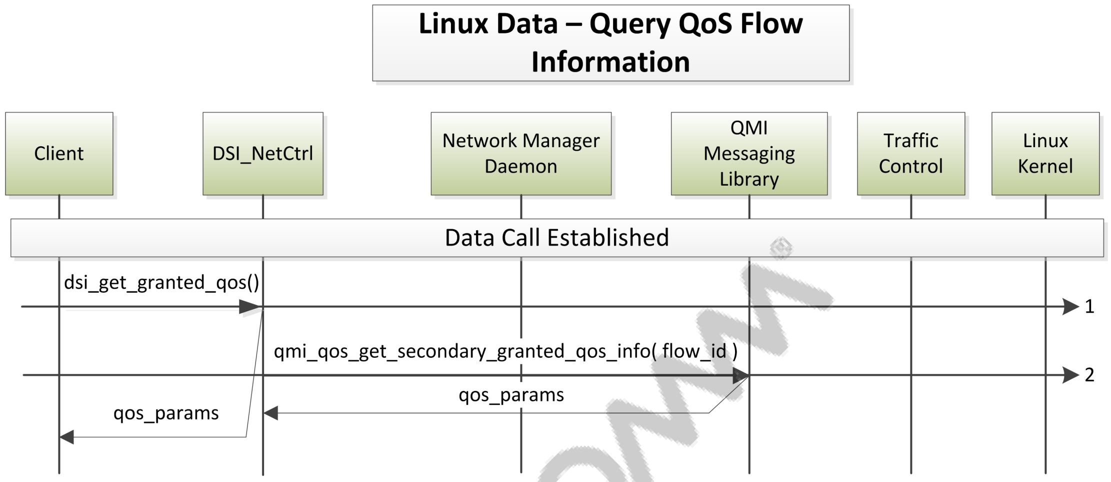

Figure 2-9 Query QoS

The client can query QoS granted information by using dsi_get_granted_qos(). dsi_get_granted_qos() returns the QoS that the network has granted for the client. Sometimes the network-granted QoS flow may be different than client-requested QoS specifications. So clients are advised to look at the results of the granted flow specification before performing any further actions.

3 Data Types

This chapter describes the DSI_NetCtrl Library data types.

- Datatypes

3.1 Datatypes

3.1.1 Define Documentation

3.1.1.1 #define DSI_SUCCESS 0

Return value from most of the DSI functions. Indicates that the operation was successful.

3.1.1.2 #define DSI_ERROR -1

Indicates that the operation was not successful.

3.1.2 Data Structure Documentation

3.1.2.1 struct dsi_call_param_value_t

Specifies string parameter values for dsi_set_data_call_param().

Data fields

Type	Parameter	Description
char *	buf_val	Pointer to the buffer containing the parameter value that is to be set.
int	num_val	Size of the parameter buffer.

3.1.2.2 union evt_info_u

Event payload sent with event callback.

Data fields

Type	Parameter	Description
struct qos_info_s	qos_info	Event information associated with a QoS status indication. See qos_info_s in the next subsection.
dsi_embms_tmgi_info_type	embms_tmgi_info	Event information associated with eMBMS event information.

3.1.2.3 struct evt_info_u::qos_info_s

Data fields

Type	Parameter	Description
dsi_qos_id_-type	flow_id	QoS flow ID .
dsi_qos_flow_-type	flow_type	Flow type.
dsi_qos_status-_event_type	status_evt	Flow status. See dsi_qos_status_event_type.
dsi_qmi_qos_-reason_code_-type	reason_code	Reason code if the flow is disconnected.

3.1.2.4 struct dsi_addr_s

Structure used to represent the IP address.

Data fields

Type	Parameter	Description
char	valid_addr	Stores whether the address is valid (for IPv6).
struct sockaddr-_storage	addr	Address structure.

3.1.2.5 struct dsi_addr_info_s

IP address information structure.

Data fields

Type	Parameter	Description
dsi_addr_t	iface_addr_s	Interface IP address.
unsigned int	iface_mask	Subnet mask.
dsi_addr_t	gtwy_addr_s	Gateway IP address.
unsigned int	gtwy_mask	Subnet mask.
dsi_addr_t	dnsp_addr_s	Primary DNS address.
dsi_addr_t	dnss_addr_s	Secondary DNS address.

3.1.2.6 struct dsi_data_pkt_stats

Packet statistics structure (returned with dsi_get_pkt_stats).

Data fields

Type	Parameter	Description
unsigned long	pkts_tx	Number of packets transmitted.
unsigned long	pkts_rx	Number of packets received.

Type	Parameter	Description
long long	bytes_tx	Number of bytes transmitted.
long long	bytes_rx	Number of bytes received.
unsigned long	pkts_dropped_-tx	Number of transmit packets dropped.
unsigned long	pkts_dropped_-rx	Number of receive packets dropped.

3.1.3 Enumeration Type Documentation

3.1.3.1 enum dsi_call_info_enum_e

Specifies which configuration parameter to update using dsi_set_data_call_param().

Enumerator:

DSI_CALL_INFO_UMTS_PROFILE_IDX UMTS profile ID.
DSI_CALL_INFO_APN_NAME APN name.
DSI_CALL_INFO_USERNAME APN user name (if any).
DSI_CALL_INFO_PASSWORD APN password (if any).
DSI_CALL_INFO_AUTH_PREF Authentication preference.
DSI_CALL_INFO_CDMA_PROFILE_IDX CDMA profile ID.
DSI_CALL_INFO_TECH_PREF Technology preference.
DSI_CALL_INFO_IP_VERSION Preferred IP family for the call.

3.1.3.2 enum dsi_net_evt_t

DSI event names.

These event names are sent along with the callback that has been registered during dsi_get_data_service_hndl().

Enumerator:

DSI_EVT_INVALID Invalid event.
DSI_EVT_NET_IS_CONN Call is connected.
DSI_EVT_NET_NO_NET Call is disconnected.
DSI_EVT_PHYSLINK_DOWN_STATE Physlink becomes dormant.
DSI_EVT_PHYSLINK_UP_STATE Physlink becomes active.
DSI_EVT_NET_RECONFIGURED Interface is reconfigured.
DSI_EVT_QOS_STATUS_IND A status for the associated QoS has changed.
DSI_EVT_NET_NEWADDR New address is generated.
DSI_EVT_NET_DELADDR An address for the interface has been deleted.
DSI_EVT_NET_PARTIAL_CONN Address is available for either IPv4 or IPv6 only.

3.1.3.3 enum dsi_data_bearer_tech_t

Bearer technology types (returned with dsi_get_current_data_bearer_tech).

Enumerator:

DSI_DATA_BEARER_TECH_UNKNOWN	Unknown bearer.
DSI_DATA_BEARER_TECH_CDMA_1X	1X technology.
DSI_DATA_BEARER_TECH_EVDO_REV0	CDMA Rev 0.
DSI_DATA_BEARER_TECH_EVDO_REVA	CDMA Rev A.
DSI_DATA_BEARER_TECH_EVDO_REVB	CDMA Rev B.
DSI_DATA_BEARER_TECH_EHRPD	EHRPD.
DSI_DATA_BEARER_TECH_FMC	Fixed mobile convergence.
DSI_DATA_BEARER_TECH_WCDMA	WCDMA.
DSI_DATA_BEARER_TECH_GPRS	GPRS.
DSI_DATA_BEARER_TECH_HSDPA	HSDPA.
DSI_DATA_BEARER_TECH_HSUPA	HSUPA.
DSI_DATA_BEARER_TECH_EDGE	EDGE.
DSI_DATA_BEARER_TECH_LTE	LTE.
DSI_DATA_BEARER_TECH_HSDPA_PLUS	HSDPA+.
DSI_DATA_BEARER_TECH_DC_HSDPA_PLUS	DC HSDPA+.
DSI_DATA_BEARER_TECH_64_QAM	64 QAM.
DSI_DATA_BEARER_TECH_TDSCDMA	TD-SCDMA.

3.1.3.4 enum dsi_qos_status_event_type

Event status types associated with QoS operations. These are included along with DSI_EVT_QOS_STATUS_IND.

Enumerator:

DSI_QOS_ACTIVATED_EV	QoS flow is activated.
DSI_QOS_SUSPENDED_EV	QoS flow is suspended.
DSI_QOS_GONE_EV	QoS flow is released.
DSI_QOS_MODIFY_ACCEPTED_EV	QoS flow modify operation is successful.
DSI_QOS_MODIFY_REJECTED_EV	QoS flow modify operation is rejected.
DSI_QOS_INFO_CODE_UPDATED_EV	New information code is available regarding the QoS status.
DSI_QOS_FLOW_ENABLED_EV	QoS data path flow is enabled.
DSI_QOS_FLOW_DISABLED_EV	QoS data path flow is disabled.

3.1.4 Typedef Documentation

3.1.4.1 typedef enum dsi_call_info_enum_e dsi_call_param_identifier_t

Specifies which configuration parameter to update using dsi_set_data_call_param().

3.1.4.2 typedef union evt_info_u dsi_evt_payload_t

Event payload sent with event callback.

3.1.4.3 typedef void(∗ dsi_net_ev_cb)(dsi_hndl_t hndl,void ∗user_data,dsi_net_evt_t evt,dsi_evt_payload_t ∗payload_ptr)

Callback function prototype for DSI NetCtrl events.

3.1.4.4 typedef struct dsi_addr_s dsi_addr_t

Structure used to represent the IP address.

3.1.4.5 typedef struct dsi_addr_info_s dsi_addr_info_t

IP address information structure.

4 Data Call Control Interfaces

This chapter describes the public DSI interfaces of the DSI_NetCtrl library.

- dsi_init
- dsi_get_data_srvc_hndl
- dsi_rel_data_srvc_hndl
- dsi_start_data_call
- dsi_stop_data_call
- dsi_set_data_call_param
- dsi_get_device_name
- dsi_get_call_end_reason
- dsi_get_call_tech
- dsi_get_ip_addr_count
- dsi_get_ip_addr
- dsi_get_current_data_bearer_tech
- dsi_reset_pkt_stats
- dsi_get_pkt_stats

4.1 dsi_init

4.1.1 Function Documentation

4.1.1.1 int dsi_init (int *mode*)

Initializes the DSI_NetCtrl library for the specified operating mode.

This function must be invoked once per process, typically on process startup.

Note: Only DSI_MODE_GENERAL is to be used by applications.

Parameters

`in`	*mode*	Mode of operation in which to initialize the library. The library can be initialized to operate in one of two modes: • DSI_MODE_GENERAL • DSI_MODE_TEST

Returns

DSI_SUCCESS – Initialization was successful.
DSI_ERROR – Initialization failed.

Dependencies

None.

4.2 dsi_get_data_srvc_hndl

4.2.1 Function Documentation

4.2.1.1 dsi_hndl_t dsi_get_data_srvc_hndl (dsi_net_ev_cb *cb_fn*, void ∗ *user_data*)

Gets an opaque data service handle. All subsequent functions use this handle as an input parameter.

Note: The DSI_NetCtrl library waits for initialization from the lower layers (QMI ports being opened, the RmNet interfaces being available, etc.) to support data services functionality. During initial bootup scenarios, these dependencies may not be available, which will cause dsi_get_data_srvc_hndl to return an error. In such cases, clients are asked to retry this function call repeatedly using a 500 ms timeout interval. Once a non-NULL handle is returned, clients can exit out of the delayed retry loop.

Parameters

`in`	*cb_fn*	Client callback function used to post event indications. Refer to Section 3.2.3 for the function prototype.
`in`	*user_data*	Pointer to the client context block (cookie). The value may be NULL.

Returns

dsi_hndl_t if successfull, NULL otherwise.

Dependencies

dsi_init() must be called.

4.3 dsi_rel_data_srvc_hndl

4.3.1 Function Documentation

4.3.1.1 void dsi_rel_data_srvc_hndl (dsi_hndl_t *hndl*)

Releases a data service handle. All resources associated with the library are released.

Note: If the user starts an interface with this handle, the corresponding interface is stopped before the DSI hndl is released.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().

Returns

None.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().

4.4 dsi_start_data_call

4.4.1 Function Documentation

4.4.1.1 int dsi_start_data_call (dsi_hndl_t *hndl*)

Starts a data call.

An immediate call return value indicates whether the request was sent successfully. The client receives asynchronous notfications via a callback registered with dsi_get_data_srvc_hndl() indicating the data call bring-up status.

Parameters

`in`	*hndl*	Handle received from dsi_get_data_srvc_hndl().

Returns

DSI_SUCCESS – The data call start request was sent successfully.
DSI_ERROR – The data call start request was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().

4.5 dsi_stop_data_call

4.5.1 Function Documentation

4.5.1.1 int dsi_stop_data_call (dsi_hndl_t *hndl*)

Stops a data call.

An immediate call return value indicates whether the request was sent successfully. The client receives asynchronous notification via a callback registered with dsi_get_data_srvc_hndl() indicating the data call tear-down status.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().

Returns

DSI_SUCCESS – The data call stop request was sent successfully.
DSI_ERROR – The data call stop request was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
The call must have been brought up using dsi_start_data_call().

4.6 dsi_set_data_call_param

4.6.1 Function Documentation

4.6.1.1 int dsi_set_data_call_param (dsi_hndl_t *hndl*, dsi_call_param_identifier_t *identifier*, dsi_call_param_value_t ∗ *info*)

Sets the data call parameter before trying to start a data call.

Clients may call this function multiple times with various types of parameters to be set.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
in	*identifier*	Identifies the type of the third input parameter.
in	*info*	Parameter value that is to be set.

Returns

DSI_SUCCESS – The data call parameter was set successfully.
DSI_ERROR – The data call parameter was not set successfully.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().

4.7 dsi_get_device_name

4.7.1 Function Documentation

4.7.1.1 int dsi_get_device_name (dsi_hndl_t *hndl,* char ∗ *buf,* int *len*)

Queries the data interface name for the data call associated with the specified data service handle.

Note: len must be at least DSI_CALL_INFO_DEVICE_NAME_MAX_LEN + 1 long.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
out	*buf*	Buffer to hold the data interface name string.
in	*len*	Length of the buffer allocated by client.

Returns

DSI_SUCCESS – The data interface name was returned successfully.
DSI_ERROR – The data interface name was not returned successfully.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().

4.8 dsi_get_call_end_reason

4.8.1 Function Documentation

4.8.1.1 int dsi_get_call_end_reason (dsi_hndl_t *hndl,* dsi_ce_reason_t ∗ *ce_reason,* dsi_ip_family_t *ipf*)

Queries for the reason for a call being ended.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
out	*ce_reason*	Structure to hold the fields of the call ending reason.
in	*ipf*	IP family for which the call end reason was requested.

Returns

DSI_SUCCESS – The call end reason was queried successfully.
DSI_ERROR – The call end reason query was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().

4.9 dsi_get_call_tech

4.9.1 Function Documentation

4.9.1.1 int dsi_get_call_tech (dsi_hndl_t *hndl,* dsi_call_tech_type ∗ *call_tech*)

Gets the techcology on which the call was brought up. This function can be called any time after the client receives the DSI_EVT_NET_IS_CONN event and before the client releases the dsi handle.

On successful return, the call_tech parameter is set to a valid call technology.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
out	*call_tech*	Pointer to the buffer containing the call technology.

Returns

DSI_SUCCESS – The call bring-up technology was queried successfully.
DSI_ERROR – The call bring-up technology query was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().

4.10 dsi_get_ip_addr_count

4.10.1 Function Documentation

4.10.1.1 unsigned int dsi_get_ip_addr_count (dsi_hndl_t *hndl*)

Gets the number of IP addresses (IPv4 and global IPv6) associated with the DSI interface.

Parameters

`in`	*hndl*	Handle received from dsi_get_data_srvc_hndl().

Returns

The number of IP addresses associated with the DSI.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl()

4.11 dsi_get_ip_addr

4.11.1 Function Documentation

4.11.1.1 int dsi_get_ip_addr (dsi_hndl_t *hndl,* dsi_addr_info_t ∗ *info_ptr,* int *len*)

Gets the IP address information structure (network order).

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
out	*info_ptr*	Pointer to the buffer containing the network order information.
in	*len*	Length of the network order information.

Returns

DSI_SUCCESS – The network order query was successful.
DSI_ERROR – The network order query was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
The length parameter can be obtained by calling dsi_get_ip_addr_count().
It is assumed that the client has allocated memory for enough structures specified by the len field.

4.12 dsi_get_current_data_bearer_tech

4.12.1 Function Documentation

4.12.1.1 dsi_data_bearer_tech_t dsi_get_current_data_bearer_tech (dsi_hndl_t *hndl*)

Returns the current data bearer technology on which a call was brought up.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().

Returns

The data bearer technology.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().

4.13 dsi_reset_pkt_stats

4.13.1 Function Documentation

4.13.1.1 int dsi_reset_pkt_stats (dsi_hndl_t *hndl*)

Resets the packet data transfer statistics.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().

Returns

DSI_SUCCESS – The packet data transfer statistics were reset successfully.
DSI_ERROR – The packet data transfer statistics reset was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().

4.14 dsi_get_pkt_stats

4.14.1 Function Documentation

4.14.1.1 int dsi_get_pkt_stats (dsi_hndl_t *hndl,* dsi_data_pkt_stats ∗ *dsi_data_stats*)

Queries the packet data transfer statistics from the current packet data session.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
in	*dsi_data_stats*	Memory to hold the queried statistics details.

Returns

DSI_SUCCESS – The packet data transfer statistics were queried successfully.
DSI_ERROR – The packet data transfer statistics query was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl()

5 QoS Interfaces

This chapter describes the public interface of the DSI_NetCtrl Library QoS.

- dsi_request_qos
- dsi_release_qos
- dsi_modify_qos
- dsi_suspend_qos
- dsi_resume_qos
- dsi_get_granted_qos
- dsi_get_qos_status

5.1 dsi_request_qos

5.1.1 Function Documentation

5.1.1.1 int dsi_request_qos (dsi_hndl_t *hndl,* unsigned int *num_qos_specs,* dsi_qos_spec_type ∗ *qos_spec_list,* dsi_qos_req_opcode_type *req_opcode,* dsi_qos_id_type ∗ *qos_id_list,* dsi_qos_err_rsp_type ∗ *qos_spec_err_list*)

Requests a new QoS flow and filter on a pre-existing network connection.

Multiple QoS specifications may be specified per invocation; however, the number of specifications supported is network service dependent. Each QoS specification contains one or more flow/filter specifications.

For network service supporting multiple flow/filter specifications (e.g., 3GPP2), only one flow/filter specification in a set is granted via negotiation with the network.

The return code indicates a successful start of the transaction by the modem, while the final outcome is determined via asynchronous event indications per Section (?∗? fill in later). Subsequent QoS status event indications are automatically generated for each flow created on the modem

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
in	*num_qos_specs*	Number of QoS specifications passed in the qos_spec_list parameter (limited to a maximum of 10). This also dictates the size of qos_id_list and qos_spec_err_list parameters, for which the client must allocate storage.
in	*qos_spec_list*	Array of QoS specifications. One or more instances of this type may be present.
in	*req_opcode*	Request operation code. Indicates a request or configure operation.
out	*qos_id_list*	List of opaque handles for each QoS flow created.
out	*qos_spec_err_list*	Array of QoS specification errors returned from the modem.

Returns

DSI_SUCCESS – The request was sent successfully.
DSI_ERROR – The request was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
There must be an active data call for the handle.

5.2 dsi_release_qos

5.2.1 Function Documentation

5.2.1.1 int dsi_release_qos (dsi_hndl_t *hndl*, unsigned int *num_qos_ids*, dsi_qos_id_type ∗ *qos_id_list*)

Deletes existing QoS flow and filters from a network connection.

The return code indicates a successful start of the transaction by the modem, while the final outcome is determined via an asynchronous event indication per Section ?∗?.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
in	*num_qos_ids*	Number of QoS flow handles passed in the qos_id_list parameter.
in	*qos_id_list*	Array of QoS flow handles. One or more instances of this type may be present.

Returns

DSI_SUCCESS – The release operation was successful.
DSI_ERROR – The release operation was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
QoS flows must be allocated using dsi_request_qos().

5.3 dsi_modify_qos

5.3.1 Function Documentation

5.3.1.1 int dsi_modify_qos (dsi_hndl_t *hndl,* unsigned int *num_qos_specs,* dsi_-qos_spec_type ∗ *qos_spec_list,* dsi_qos_err_rsp_type ∗ *qos_spec_err_list*)

Changes existing QoS flows on a network connection.

The return code indicates a successful start of the transaction by the modem, while the final outcome is determined via an asynchronous event indication per Section ?∗?.

Parameters

`in`	*hndl*	Handle received from dsi_get_data_srvc_hndl().
`in`	*num_qos_specs*	Number of QoS specifications passed in the qos_spec_list parameter. This also dictates the size of the qos_id_list and qos_spec_err_list parameters, for which the client must allocate storage.
`in`	*qos_spec_list*	Array of QoS specifications. One or more instances of this type may be present.
`out`	*qos_spec_err_list*	Array of QoS specification errors returned from the modem.

Returns

DSI_SUCCESS – The modify operation was successful.
DSI_ERROR – The modify operation was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
QoS flows must be allocated using dsi_request_qos().

5.4 dsi_suspend_qos

5.4.1 Function Documentation

5.4.1.1 int dsi_suspend_qos (dsi_hndl_t *hndl,* unsigned int *num_qos_ids,* dsi_qos_id_type ∗ *qos_id_list*)

Disables prioritized packet handling for existing QoS flows on a network connection. Further packet transmission on those QoS flows is treated as best-effort traffic.

The return code indicates a successful start of the transaction by the modem, while the final outcome is determined via an asynchronous event indication per Section ?∗?.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
in	*num_qos_ids*	Number of QoS flow handles passed in the qos_id_list parameter.
in	*qos_id_list*	Array of QoS flow handles. One or more instances of this type may be present.

Returns

DSI_SUCCESS – The suspend operation was successful.
DSI_ERROR – The suspend operation was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
QoS flows must be allocated using dsi_request_qos().

5.5 dsi_resume_qos

5.5.1 Function Documentation

5.5.1.1 int dsi_resume_qos (dsi_hndl_t *hndl,* unsigned int *num_qos_ids,* dsi_qos_id_type ∗ *qos_id_list*)

Enables prioritized packet handling for existing QoS flows on a network connection.

The return code indicates a successful start of the transaction by the modem, while the final outcome is determined via an asynchronous event indication per Section ?∗?.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
in	*num_qos_ids*	Number of QoS flow handles passed in the qos_id_list parameter.
in	*qos_id_list*	Array of QoS flow handles. One or more instances of this type may be present.

Returns

DSI_SUCCESS – The resume operation was successful.
DSI_ERROR – The resume operation was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
QoS flows must be allocated using dsi_request_qos().
Only a suspended QoS flow can be resumed.

5.6 dsi_get_granted_qos

5.6.1 Function Documentation

5.6.1.1 int dsi_get_granted_qos (dsi_hndl_t *hndl,* dsi_qos_id_type *qos_id,* dsi_qos_granted_info_type ∗ *qos_info*)

Queries the QoS information for an existing flow on a network connection.

Parameters

`in`	*hndl*	Handle received from dsi_get_data_srvc_hndl().
`in`	*qos_id*	QoS flow handle.
`out`	*qos_info*	QoS information returned from the modem.

Returns

DSI_SUCCESS – The query was successful.
DSI_ERROR – The query operation was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
QoS flows must be allocated using dsi_request_qos().

5.7 dsi_get_qos_status

5.7.1 Function Documentation

5.7.1.1 int dsi_get_qos_status (dsi_hndl_t *hndl,* dsi_qos_id_type *qos_id,* dsi_qos_status_type ∗ *qos_status*)

Queries the QoS activated/suspended/gone state for an existing flow on a network connection.

Parameters

in	*hndl*	Handle received from dsi_get_data_srvc_hndl().
in	*qos_id*	QoS flow handle.
out	*qos_status*	QoS flow status returned from the modem.

Returns

DSI_SUCCESS – The query was successful.
DSI_ERROR – The query operation was unsuccessful.

Dependencies

dsi_init() must be called.
The handle must be a valid handle obtained by dsi_get_data_srvc_hndl().
QoS flows must be allocated using dsi_request_qos().