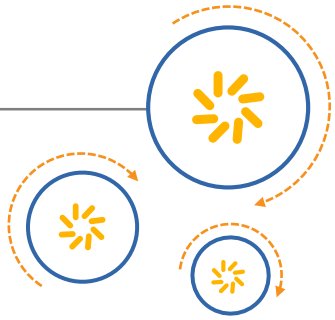
Qualcomm Technologies, Inc.



Data Services API Interface Specification

80-V6415-1 YD

July 17, 2015

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

Revision history

Revision	Date	Description
A	March 2003	Initial release
B	June 2003	Engineering updates
C	Sep 2003	▪ Added dss_open_netlib2(), dss_init_net_policy_info(), dss_set_app_net_policy(), dss_get_app_net_policy() ▪ Modified dss_get_iface_id() and made other minor changes
D	Dec 2003	Added dss_get_iface_id_by_policy() and made other minor changes
E	Mar 2004	▪ Added Error Notification, Socket Parameter Tuning, dsnet_get_handle(), dss_socket2(), dssnet_get_policy(), dsnet_set_policy() ▪ Modified dss_get_iface_id_by_policy(), dss_get_iface_id(), and made other minor changes
F	June 2004	▪ Added Section 3.8.1 ▪ Updated dss_setsockopt() and dss_getsockopt()
G	Nov 2004	Added Chapter 6
H	Dec 2004	▪ Updated DSS_IFACE_IOCTL_GO_ACTIVE, DSS_IFACE_IOCTL_GO_DORMANT description. ▪ Added DSS_IFACE_IOCTL_MT_REG_CB, DSS_IFACE_IOCTL_MT_DEREG_CB, DSS_IFACE_IOCTL_QOS_AVAILABLE_DEACTIVATED_EV ▪ Updated DSS_GET_INTERFACE_ID description
J	Mar 2005	Added changes associated with DSS_VERSION 900
K	Aug 2005	Added changes associated with DSS_VERSION 1000
L	Oct 2005	Engineering updates
M	Dec 2005	Engineering updates
N	Sep 2006	Engineering updates
P	Dec 2006	Engineering updates
R	Apr 2007	▪ Updated list of interface control events ▪ Updated DSS_SO_SBD_ACK_CB ▪ Added CSS_SO_DISABLE_FLOW_FWDING socket option ▪ Added Ping section
T	Sep 2007	Engineering updates
U	May 2008	Engineering updates
V	Dec 2009	Engineering updates
W	Feb 2011	Updated Sections 3.1.5 and 3.2.1; updated Table 2-3 and Table 4-3; added Section 4.2.4.6
Y	Dec 2011	Engineering updates, including the addition of Sections 4.2.4.2.3, 4.2.4.2.4, 4.2.4.2.6, 4.2.4.2.18, 6.2.2.11, and 6.3.3.3
YA	Jun 2013	Updated References section: Removed original Q5 and Q6 references (Rev Y) and changed Q7 to Q5 (also links)
YB	Jul 2013	Removed Q5 reference
YC	May 2014	Added event to Table 2.4; added socket options to section 3.8.1 and Table 3.-2; updated section 4.1.8
YD	Jul 2015	Engineering updates on pages 66, 89, 96, 97, 179, 184, 185, 191 (shaded)

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document is the reference specification for the Data Services (DS) Application Programming Interface (API). This API is designed to facilitate the development of mobile station-based networking applications.

1.2 Scope

This document specifies the Data Services API that network applications will use to communicate with the CDMA Data Services protocol stack on a multimode Mobile Station Modem (MSM™) ASIC. The Data Services API includes programming interfaces to access the TCP/IP protocol stack, the network subsystem, and the link layer interfaces capable of transporting IP packets.

The Data Services API addresses a limited set of requirements to support packet-based applications running on the mobile station. However, the API can be readily extended as long as any extension maintains compatibility with this API so that the applications do have to be rewritten.

This document assumes that the audience is familiar with the Dual-Mode Subscriber Station (DMSS) software environment and has knowledge of QUALCOMM® Incorporated's Real-Time Executive (REX) operating system and its task operation. In addition, a rudimentary knowledge of sockets programming is assumed.

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 and command variables appear in a different font. E.g., **`copy a:*.* b:`**.

Parameter types are indicated by arrows:

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

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

1.4 Technical assistance

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

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

2 Operation

A well-defined API can greatly facilitate the development of network applications. Traditional network programming makes use of the Sockets API to communicate with the underlying communication protocols. Berkeley Software Distribution (BSD) sockets and Winsock™ are two commonly used socket APIs.

The CDMA Data Services protocol stack provides the DS Sockets API and the Network Subsystem and Interface Control API to support mobile applications that need to use the data protocol stack. Figure 2-1 illustrates the placement of these APIs with respect to the protocol stack. These two APIs are collectively referred to as the Data Services API.

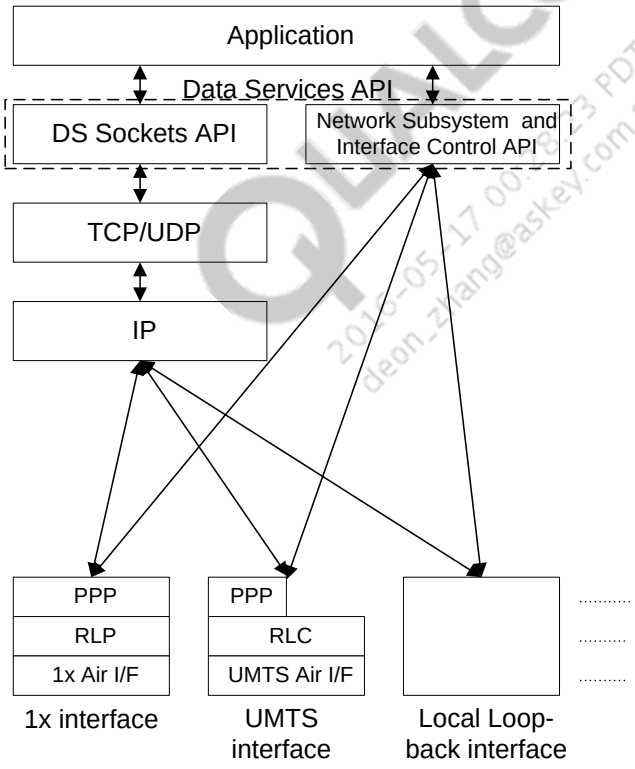


Figure 2-1 CDMA Data Services stack with the Data Services API

The DS Sockets API provides an interface to the transport layer, i.e., Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). This includes the traditional BSD system calls to set up a client or server socket, transfer data, perform administrative operations, and to terminate the connection. The DS Sockets API also supports DNS look-up calls and provides several utility functions.

The Network Subsystem and Interface Control API abstract the technology-specific details from the protocol stack and the application. This enables the application and the protocol stack to transparently run over any link-layer technology. The Network Subsystem API deals with bringing up the network subsystem, tearing it down, providing its status, and performing other network subsystem-specific operations. The Interface Control API provides an abstraction for the lower layers, allowing the addition of new link-layer technologies without requiring any modifications to the applications or the protocol stack. The Interface Control functions enable applications to retrieve and configure attributes, perform control operations, and receive notifications of interface-specific events.

Both of these APIs are nonblocking and use event notification mechanisms with callbacks to notify the application of events of interest.

2.1 Requirements

The Data Services API requirements are as follows:

- Any enhancements to the API should retain this implementation as a proper subset so that applications written for this implementation will work with future sockets interfaces with minimal modification to the application.
- The API works only under REX.
- All communication between the API and the application is performed through function calls.

2.2 Constraints

The constraints of the Data Services API are as follows:

- This API supports the local loop-back sockets and sockets over the air interface. It does not support sockets over the R_m interface.
- The implementation supports only TCP and UDP sockets. It does not support raw IP or Internet Control Message Protocol (ICMP) sockets.
- The implementation supports multiple TCP and UDP sockets. The number of concurrent TCP and UDP sockets is determined at compile time based on DSS_MAX_TCP_SOCKS and DSS_MAX_UDP_SOCKS defined in dssocket.h.
- The API only supports nonblocking INPUT/OUTPUT. Blocking behavior would need to be emulated at the application level.
- The TCP implementation of the DMSS Data Services uses Nagle's algorithm to reduce the number of outstanding, small segments.
- Most of the TCP parameters like Maximum Segment Size (MSS), initial Round Trip Time (RTT) estimate, etc., are fixed. However, socket options are provided to modify some of them.

2.3 Usage

Header files

Applications need to include only dssocket.h, except for the DNS and address conversion routines, which are declared in dssdns.h.

Featurization

Use FEATURE_DS_SOCKETS to enable the DS Sockets API and Network Subsystem and Interface Control API.

REX

If the application is running under REX, it is preferable that applications maintain a separate REX task outside of the Protocol Service (PS) task.

Others

The event callback functions should be used only for signaling, as they are called in the context of the PS task (or ds_sig task in newer versions).

2.4 Theory of operation

The DS API functions are called in the context of the application task. However, the actual processing happens in the context of the PS task for the majority of the functions. Processing for some of the functions is done in the context of the application task itself. The application writer must keep this in mind when deciding the priority of the application task.

The architecture supports only nonblocking INPUT/OUTPUT behavior. The application provides different callback functions to receive asynchronous notification of various events. These callback functions are socket event callback, network event callback, and various interface control-specific event callbacks. These callback functions are called in the context of the PS task to notify the application of events of interest (or ds_sig task in newer versions). These callback functions are called with the task switch disabled, so the application should use its native method in the callback function to communicate the event to the application task, i.e., to generate a signal, send a message, assert an interrupt, etc. Any other code in the callback function would result in a task switch being disabled for a longer time. We also observe that one can write a layer within the application context to emulate blocking INPUT/OUTPUT, if necessary.

Typically, an application needs a network handle, also referred to as an application handle, before it can use the DS API. An exception to this rule is the dss_iface_ioctl() call, which may be called without having obtained the network handle in some situations. The application needs to call dss_open_netlib() to get the network handle. This call requires a network handle to pass a socket callback and a Network Callback function. Essentially, the dss_open_netlib() call links the DS Sockets API with the network subsystem. The Socket Callback function is called to inform the application of the socket-specific events, e.g., the socket is writeable, and the network callback is called to inform the application of network-specific events, e.g., the network came up.

After getting the application handle, the application typically brings up the network subsystem by calling dsnet_start(). This call is responsible for bringing up both the traffic channel and the link layer. Note that this is an application-wide call and does not depend on an individual socket. The network callback is invoked upon successful establishment or failure of the network subsystem, and the application uses the dss_netstatus() call to query the network state.

A socket is created through a call to dss_socket(), which allocates the necessary internal resources and provides a socket descriptor to the caller. After the underlying protocols are established with a dsnet_start() call, one can establish a TCP connection by calling dss_connect() in case of a client socket and dss_accept() in case of a server socket and then transferring data using dss_read() and dss_write() calls. In case of UDP sockets, one can directly call dss_sendto() or dss_recvfrom() to perform data transfer. The DS Sockets API also supports other standard socket API functions. See Chapter 3 for complete details.

Upon completion of the data transfer, the opened sockets are closed by calling dss_close() and the network subsystem closed by calling dsnet_stop(). The resources allocated to the application also need to be freed up by calling dss_close_netlib().

An application may also use the Interface Control API to perform an operation on an IP interface, i.e., the application may choose to receive notifications of the IP address change of the interface, or it may force dormancy on a 1X interface after it is done with the data transfer.

The DS Sockets API, the network subsystem, and the Interface Control API define the following:

- Socket events
- Network events
- Interface control-specific events
- Registering mechanisms
- Receiving corresponding notifications

See Section 2.5.1 for DS Sockets API-related event management information and Section 2.6.1 for the Network Subsystem and Interface Control API-related event management information.

2.5 DS Sockets API

The DS Sockets API is similar to the standard sockets API available on the UNIX platforms. The significant difference between the two is that the DS Sockets API supports only nonblocking behavior and leaves it up to the application to emulate a blocking behavior, if desired. In addition, the DS Sockets API uses event notification callbacks to support the asynchronous, nonblocking behavior of the calls.

The constant DSS_VERSION, defined in dssocket.h, specifies the version number of the DS Sockets API. The version number constitutes a major and minor version number. The least significant digit is the minor version number, while the most significant digits form the major version number. The version number is incremented when the behavior of an existing function is changed or when new functions or options are added to the API. The version number can be used by applications to maintain backward compatibility.

2.5.1 Event notifications

The application may register to receive asynchronous notification of the socket events of interest through the dss_async_select() function. The application specifies the socket identifier and a bit mask (multiple events ORed together) of the socket events (referred to as an event mask) for which it wants to receive notification. The dss_async_select() function performs a real-time check to determine if one or more of the specified events have already occurred. If so, it calls the socket callback function, registered as part of the dss_open_netlib() call, immediately. Otherwise, the callback function is called whenever any of the specified events is asserted.

After the application has been notified of the event through the socket event callback, it must call dss_getnextevent() to determine which events have occurred. dss_getnextevent() provides real-time checking of which events occurred and returns a signed 31-bit event mask indicating which events have occurred. In addition, the corresponding bits associated with these events are cleared internally. The application will no longer receive notification of these events, and the application must call dss_async_select() again to register.

NOTE: A signed quantity is required to allow a negative return value on error.

The application may deregister the events of its interest by calling dss_async_deselect(). This function clears the events, specified through a bit mask, from the interest mask associated with the socket. Table 2-1 provides a list of the socket events supported by this architecture.

Table 2-1 List of DS socket API events

Event	Description
DS_READ_EVENT	Indicates that a call to an input routine would return a value indicating something other than the operation would block (DS_EWOULDBLOCK)
DS_WRITE_EVENT	Indicates that a call to an output routine would return a value indicating something other than the operation would block (DS_EWOULDBLOCK)
DS_CLOSE_EVENT	Indicates that a call to the close routine would return a value indicating something other than the operation would block (DS_EWOULDBLOCK)
DS_ACCEPT_EVENT	Indicates that a call to the accept routine would return a value indicating something other than the operation would block (DS_EWOULDBLOCK)

2.5.2 Error notification

Applications are informed about asynchronous errors either through socket events notifications or in errno parameters when input/output functions are called after the event.

If a TCP connection establishment fails, or if a TCP connection is reset by peer, DS_READ_EVENT, DS_WRITE_EVENT, and DS_CLOSE_EVENT are all asserted. At this point, a call to input/output routines would result in an error with an errno value, indicating that the socket is not connected.

DS_READ_EVENT and DS_WRITE_EVENT are also asserted, in case the socket receives an ICMP error and either the DSS_SO_ERROR_ENABLE or DSS_IP_RECVERR socket option is enabled. At this point, a call to input/output routines would result in an error with an errno value, indicating the pending ICMP error on the socket.

In the case of a network subsystem failure or application-initiated network close, all the connected sockets (UDP and TCP) bound to that interface are disconnected and all three events are asserted, notifying the application of a failure.

In the case when an IP gateway handoff results in an IP address change, all the connected TCP sockets bound to that interface are disconnected and all three events are asserted.

A call to dss_close() after an error would free up the socket and make it immediately available for reuse.

2.5.3 Error handling

The DS Sockets API functions return errno values through a function call parameter. The errno variable is passed by reference as an argument, and the DS Sockets API fills in the appropriate value:

- A return value of DSS_ERROR from a function indicates that a valid value is returned in errno and the application should check and handle this error accordingly.
- A return value of DSS_SUCCESS indicates that no errno condition was reported.

Table 2-2 is a list of possible socket-based error values supported by this implementation. The socket-based calls return a generic error, DS_ENETDOWN, for errors related to the network subsystem, and an application should always handle it, as it encapsulates all unknown network subsystem errors. The application may also query the actual network subsystem state through dss_netstatus(), if necessary.

Table 2-2 DS socket error numbers

Error number value	Description
DS_EADDRINUSE	Address is already in use
DS_EADDRREQ	Destination address requested
DS_EAFNOSUPPORT	Specified address family is not supported
DS_EBADAPP	Invalid network handle was specified
DS_EBADF	Invalid socket descriptor was specified
DS_ECONNABORTED	TCP connection was aborted due to timeout or other reason
DS_ECONNREFUSED	Connection attempt was refused due to the receipt of a server RST
DS_ECONNRESET	Server reset the connection; receipt of a server RST

Error number value	Description
DS_EEOF	End of file received
DS_EFAULT	Bad buffer address; alternatively, invalid size specified for address length, message length, etc.
DS_EINPROGRESS	Connection establishment in progress
DS_EINVAL	Invalid operation
DS_EIPADDRCHANGED	As part of handoff, the network-level IP address changed, causing the TCP connection to reset
DS_EISCONN	Connection established
DS_EMAPP	Attempt to open more than one application; no additional resources available for use
DS_EMFILE	Attempt to open more than one socket descriptor; no additional resources available for use
DS_EMSGSIZE	Message is too large to be sent all at once
DS_ENOPROTOOPT	Protocol not available
DS_EMSGTRUNC	Message truncated because the supplied buffer is too small
DS_ENETDOWN	Generic network error; underlying network is not available
DS_ENOMEM	No memory available to complete the operation
DS_ENOROUTE	No route to destination found
DS_ENOTCONN	TCP connection does not exist due to lack of origination attempt, or the connection attempt failed
DS_EOPNOTSUPP	Invalid or unsupported operation was specified
DS_EPIPE	Peer closed the TCP connection
DS_EPROTONOSUPPORT	Specified protocol is not supported
DS_EPROTOTYPE	Specified protocol is the wrong type for this socket
DS_ESHUTDOWN	Write attempted after the write-half is closed
DS_ESOCKNOSUPPORT	Invalid or unsupported socket parameter was specified
DS_ETIMEDOUT	Connection establishment timed out
DS_EWOULDBLOCK	If this were a blocking function call, the operation would block
DS_SOCKEXIST	Attempt to exit the network library with a remaining socket allocated
DS_ENETUNREACH	Network is unreachable
DS_EHOSTUNREACH	No route to host
DS_EHOSTDOWN	Host is down
DS_ENONET	Host is not on network
DS_EPROTO	Protocol error
DS_EACCES	Permission denied

2.5.4 Socket parameter tuning

Various socket parameters control the resources allocated to the sockets/TCP/UDP layer. These parameters are defined as constants in header files and can be configured to suit particular requirements and configurations. The following is a brief description of various parameters and guidelines in choosing values for them:

- The implementation supports multiple TCP and UDP sockets. The number of concurrent TCP and UDP sockets is determined at compile time based on DSS_MAX_TCP_SOCKS and DSS_MAX_UDP_SOCKS defined in dssocket.h.
- The default size of the TCP/UDP Tx queue is defined by the constant DSS_DEF_SNDBUF. This value should be set based on the expected Reverse Link (RL) throughput and RTT estimate of the RL. Each socket can change its send queue size using a socket option.
- The default size of the TCP/UDP Rx queue is defined by the constant DSS_DEF_RCVBUF. This value should be based on the expected Forward Link (FL) throughput and RTT estimate of the FL. Each socket can change its Rx queue size using a socket option.
- The constants DSMI_DS_LARGE_ITEM_SIZ and DSMI_DS_SMALL_ITEM_SIZ defined in dsm.c control the memory availability for the socket's Tx and Rx queues. These values should be configured based on default sndbuf and rcvbuf values and the number of simultaneous socket connections that need to be supported.

2.6 Network Subsystem and Interface Control API

The Network Subsystem API provides calls to the application to open and close the network subsystem and perform other operations specific to the network subsystem. This API provides a logical representation of the network subsystem to each application based on an application handle. Hence, each application gets the impression that it has exclusive access to the network subsystem, e.g., when bringing up the network subsystem, it is transparent to the application whether it resulted in bringing up the physical interface or another application had already brought it up earlier.

An interface is defined as any link layer capable of transporting IP packets, and the Interface Control API provides a set of operations that can be performed on a particular interface. These operations include the ability to retrieve certain attributes (IP address, the state of an interface, etc.) of the interface as well as the ability to configure certain attributes, e.g., setting RLP NAK parameters in case of a 1X data call. This API also allows the application to register callbacks for specific events, e.g., IP address change.

2.6.1 Event notifications

The application is notified of the network events through the network callback registered as part of a dss_open_netlib() call. This event is a generic event indicating a change in the state of the network subsystem. The application should query the state of the network subsystem by calling dss_netstatus() upon receiving this event notification.

The interface control mechanism allows an application to register a callback for specific events using the dss_iface_ioctl() call. The registered callback is called when the specified event occurs. Table 2-3 provides a list of all the interface control-related events.

Table 2-3 List of the interface control events

Event	Description
DSS_IFACE_IOCTL_DOWN_EV	Indicates that the interface is down
DSS_IFACE_IOCTL_UP_EV	Indicates that the interface is up
DSS_IFACE_IOCTL_PHYS_LINK_DOWN_EV	Indicates that the physical link of the interface is down

Event	Description
DSS_IFACE_IOCTL_PHYS_ LINK_UP_EV	Indicates that the physical link of the interface is up
DSS_IFACE_IOCTL_ADDR_ CHANGED_EV	Indicates that the IP address of the interface has changed
DSS_IFACE_IOCTL_ BEARER_TECH_ CHANGED_EV	Indicates that the underlying bearer technology for the interface has changed
DSS_IFACE_IOCTL_707_ NETWORK_SUPPORTED_ QOS_PROFILES_CHANGED _EV	Indicates that the QoS profile set that is currently supported by the network has changed; the application may query for the new set using DSS_IFACE_IOCTL_GET_NETWORK_ SUPPORTED_QOS_PROFILES IOCTL.
DSS_IFACE_IOCTL_ EXTENDED_IP_CONFIG_EV	Indicates that the result of the DHCP refresh request is available; the event information returned with this event is a Boolean indicating whether or not the refresh was successful or not.

2.6.2 Error handling

The Network Subsystem and Interface Control API follow the same mechanism for error handling as the DS Sockets API. Table 2-4 provides a list of the error numbers used by the network subsystem functions. The Interface Control functions use the generic error numbers listed in Table 2-4.

Table 2-4 Network subsystem error numbers

Event	Description
DS_ ENETCLOSEINPROGRESS	Network subsystem termination is currently in progress
DS_ENETEXIST	Attempt to exit the network library while network subsystem still established
DS_ENETINPROGRESS	Network subsystem establishment is currently in progress
DS_ENETISCONN	Network subsystem established and available
DS_ENETNONET	Underlying network is unavailable
DS_ENETGOINGDORMANT	Traffic channel is going dormant

2.7 Example scenarios

The following examples further illustrate the use of this asynchronous messaging mechanism. Figure 2-2 illustrates the asynchronous connection establishment for a TCP client socket.

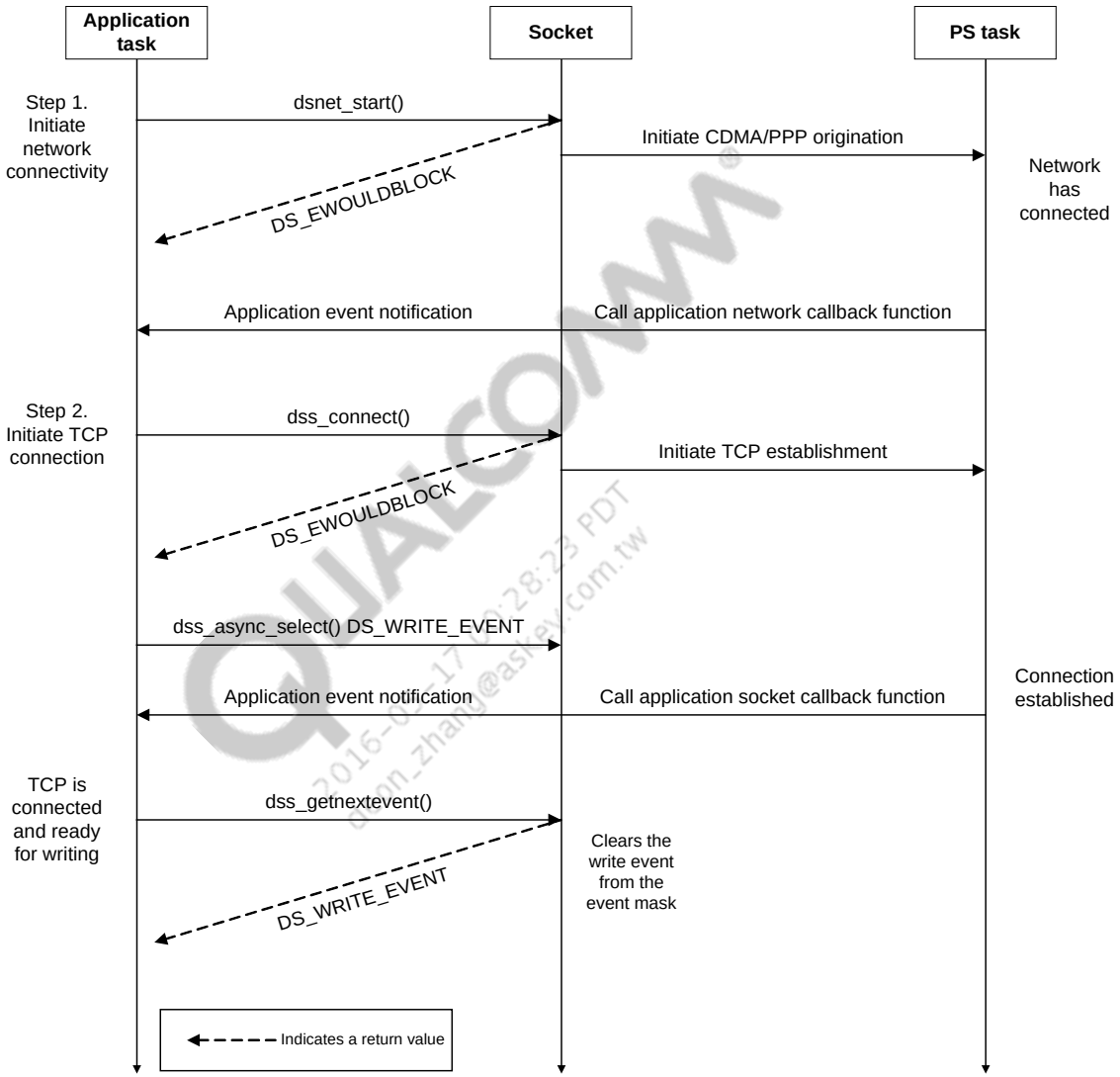


Figure 2-2 Asynchronous connection establishment for a TCP client socket

An application obtains the application handle by calling dss_open_netlib() followed by a call to dss_socket() to obtain a socket descriptor. It then initiates the asynchronous connection establishment through a call to dsnet_start(), followed by a call to dss_connect().

Steps as illustrated in Figure 2-2:

1. The initial call dsnet_start() is used to bring up the network subsystem.
   a. Since the call is nonblocking, it would immediately return with DS_EWOULDBLOCK.
   b. After the network subsystem has been established, the PS task notifies the application that the underlying network is now available through the application's network callback function.

2. The socket-based transport-level connection may then be brought up using dss_connect().
   a. Since the socket is nonblocking, the call to dss_connect() would return immediately with DS_EWOULDBLOCK, indicating that the operation would block. At this time, the application should specify events for which it wishes to receive notification. In this case, the event of interest is when the socket becomes writable, indicating completion of the connection establishment. The application specifies this through a call to dss_async_select(), which will set the corresponding bit in the interest mask internally stored with the socket, as well as perform a real-time check to determine if the event has already occurred. If no events have occurred, the function simply returns without calling any callback functions.
   b. After the transport protocol has been established, the PS task notifies the application that a connection has been established.
      i The PS task first checks to see if the application is interested in receiving a notification for this event, i.e., the bit is set in the socket's interest mask.
      ii If the event has been set in the event mask, the PS task notifies the application through the application's socket callback function.
      iii After the application has been notified that an event of interest has been asserted, it must make a call to dss_getnextevent() to determine which event occurred.
      iv Upon successfully establishing a connection, the DS_WRITE_EVENT would have been asserted, and the event would be cleared from the socket's interest mask. Subsequent notification for this particular event must be re-enabled through another call to dss_async_select().

Socket input

Figure 2-3 illustrates the sequence involved in the nonblocking socket read, assuming that the socket has established a connection with the peer. The dashed lines in the figure indicate a return value.

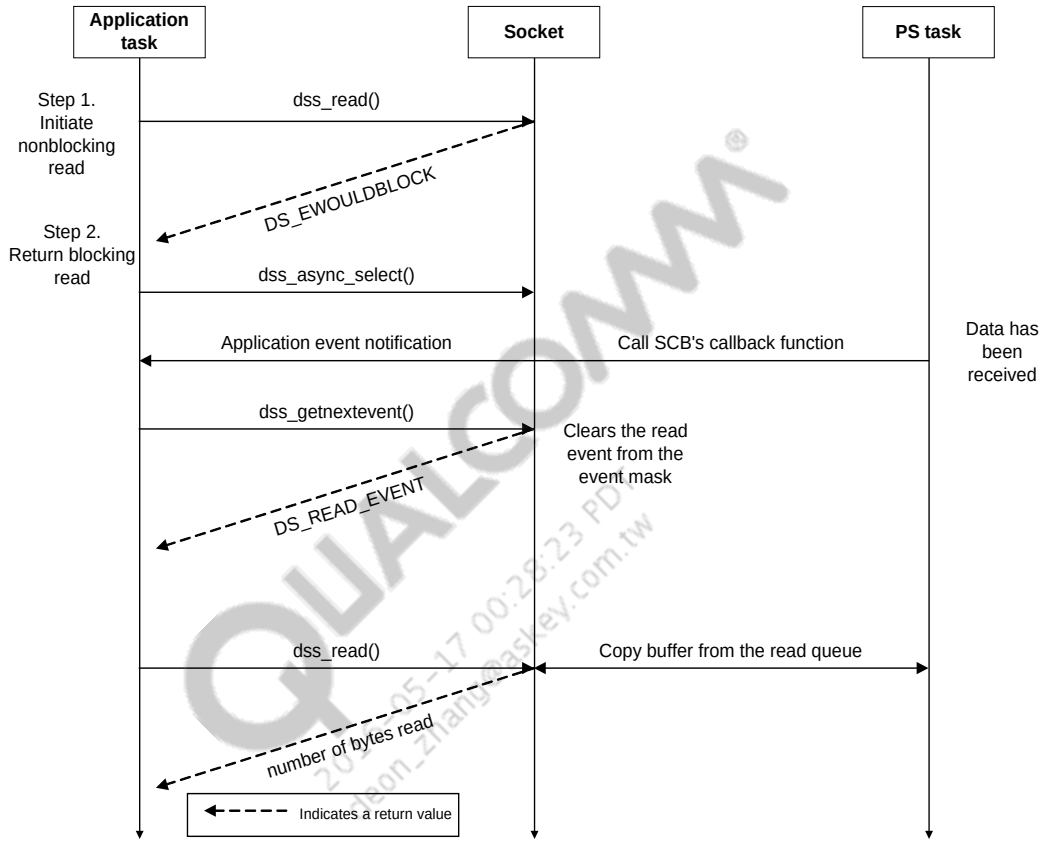


Figure 2-3 Asynchronous socket input

Steps as illustrated in Figure 2-3:

1. To initiate a nonblocking read, the application would make a call to dss_read().
2. Assuming that there is nothing available for reading, the function immediately returns with DS_EWOULDBLOCK, indicating that the operation would block.
   a. At such time, the application should specify events for which it wishes to receive notification. In this case, the socket event of interest is that the socket becomes readable, i.e., asserts DS_READ_EVENT, indicating that data is available for reading. The application specifies this through a call to dss_async_select(), which will set the corresponding bit in the socket's interest mask, as well as perform a real-time check to determine if the event has already occurred.
   b. Assuming that the event had not occurred, the call would simply return.
   c. Later, when data arrives in the Rx queue, the PS task would notify the application that data is ready for reading.
      i The PS task first checks to see if the application is interested in receiving notification of the DS_READ_EVENT.

ii If the event is set in the interest mask, the PS task notifies the application through the application's Socket Callback function.

iii Once the application has been notified that an event of interest has been asserted, it must make a call to dss_getnextevent() to determine which event occurred.

If data was available for reading, the DS_READ_EVENT would have been set, and the event is cleared from the socket's interest mask. Subsequent notification for that particular event must be re-enabled through another call to dss_async_select().

3 DS Sockets API

This chapter provides specification and example usage for the DS Sockets API functions. The specification is grouped into various sections based on the functionality, e.g., input calls are grouped together. Table 3-1 lists all the functions provided by the DS Sockets API.

Table 3-1 Socket API functions

Function	Description
dss_accept()	Accepts a new passively opened TCP connection and does not support UDP
dss_async_deselect()	Clears specific events from the interest mask for a particular socket
dss_async_select()	Specifies which events to wait on for a particular socket
dss_bind()	For all client sockets, attaches a local address and port value to the socket
dss_close()	Closes a socket, freeing it for reuse; for TCP, it is implemented as a nonblocking call
dss_close_netlib()	Closes the network library
dss_connect()	Initiates an active open for TCP connection; for UDP sockets, dss_connect sets the peer's IP address and port value of the socket
dss_dns_create_session()	Creates a session for performing DNS lookup
dss_dns_set_config_params()	Sets configuration parameters in the DNS session
dss_dns_get_config_params()	Gets configuration parameters from the DNS session
dss_dns_delete_session()	Deletes a DNS session
dss_dns_get_addrinfo()	Maps a host name to its associated socket address
dss_dns_read_addrinfo()	Reads the results of dss_dns_get_addrinfo()
dss_dns_get_nameinfo()	Maps a socket address to its associated domain name
dss_dns_read_nameinfo()	Reads the results of dss_dns_get_nameinfo()
dss_freehostent()	Frees the dss_hostent structure returned by dss_getipnodebyname() or dss_getipnodebyaddr()
dss_getipnodebyaddr()	Maps an IP address to its associated domain names
dss_getipnodebyname()	Maps a domain name to its associated IP addresses
dss_getnextevent()	Gets the next socket descriptor and events that have occurred
dss_getpeername()	Gets the address of the peer connected to the specified socket
dss_getsockname()	Gets the local address assigned to the specified socket
dss_getsockopt()	Gets the options associated with the specified socket
dss_htonl()	Converts long integers from host to network byte order
dss_htons()	Converts short integers from host to network byte order

Function	Description
dss_inet_aton()	Converts an IPv4 address from dotted string form to a 4-byte host address form
dss_inet_ntoa()	Converts an IPv4 address from 4-byte host address form to dotted string form
dss_listen()	Initiates a passive open for a TCP connection; does not support UDP
dss_ntohl()	Converts long integers from network to host byte order
dss_ntohs()	Converts short integers from network to host byte order
dss_open_netlib()	Opens the network library
dsnet_get_handle()	Opens the network library and sets the application's network policy
dss_read()	Reads specified number of bytes from the TCP/UDP transport
dss_readv()	Reads specified number of bytes from the TCP/UDP transport; provides scatter-read variant that allows reading into noncontiguous buffers
dss_recvfrom()	Reads specified number of bytes from the UDP transport
dss_sendto()	Sends specified number of bytes across the UDP transport
dss_setsockopt()	Sets the specified socket options
dss_shutdown()	Shuts down the specified socket in one or both directions
dss_socket()	Creates a socket and returns a socket descriptor
dss_write()	Sends specified number of bytes across the TCP/UDP transport
dss_writev()	Sends specified number of bytes across the TCP/UDP transport; provides gather-write variant that allows writing from noncontiguous buffers

3.1 Setup

3.1.1 dss_open_netlib()

This function opens the network library, sets the application task's network and socket callback functions, and binds them to the application control block.

Parameters

```
sint15 dss_open_netlib (
                        void     (*net_callback_fcn) (void)
                        void     void (*socket_callback_fcn) ( void*)
                        sint 15  *errno
                        )
```

→	dss_open_netlib		
	→	*net_callback_fcn	Pointer to the network callback function; invoked to inform the application whenever network-related events have occurred
	→	*socket_callback_fcn	Pointer to the socket callback function; invoked to inform the application whenever socket-related events of interest have occurred
	←	*errno	Pointer to the returned error value that is passed by reference; valid values for errno are ▪ DS_EMAPP – No more applications available; maximum number of open applications exceeded

Return value

This function returns:

- Network handle on success
- On error, returns DSS_ERROR and places the error condition value in *errno

Dependencies

None

Description

This function opens up the network library, assigns the network handle, and sets the application-defined callback functions to be called when the network and socket calls make progress. The dss_open_netlib() is essentially a blocking call, i.e., there are no events associated with or signals set when the library is opened.

This function is called from the context of the socket client's task.

Example

An example of this network callback for REX is:

```
void app_dss_net_cb
(
   void* arg1         /*dummy arg; we don't use it*/
)
{
   (void)rex_set_sigs(&app_tcb,APP_NETWORK_CB_SIG);
}
```

Where:

- APP_NETWORK_CB_SIG is the REX signal that is used for the asynchronous notification for network-related events
- app_tcb specifies the application's task control block

An example of this socket callback for REX is:

```
void app_dss_socket_cb
(
   void* arg1    /*dummy arg; we don't use it*/
)
{
   (void)rex_set_sigs(&app_tcb,APP_SOCKET_CB_SIG);
}
```

Where:

- APP_SOCKET_CB_SIG is the REX signal that is used for the asynchronous notification for socket-related events
- app_tcb specifies the application's task control block

An example usage of this function call is:

```
/*-----------------------------------------------------------------
Initializes the network library. This call registers the
currently executing task with the sockets library. Note
that only one task can have a socket open (only one TCP socket
is allowed in the system), so if competing tasks were trying
this, only one would succeed.

The application passes two callback functions. The first is used
by the sockets code when we make a sockets network library call.
The second is used when we make a sockets library call.
-----------------------------------------------------------------*/
MSG_MED("Opening sockets library",0,0,0);
socket_nethandle = dss_open_netlib(&app_dss_net_cb,
                                   &app_dss_socket_cb,
                                   &error_num
```

```
                                    );
    if (socket_nethandle == DSS_ERROR)
    {
         /* We have an error. We can look at error_num to see what
          * it was and put value to DIAG.
          */

        ERR("Could not open network library",0,0,0);
        return;
    }

    MSG_MED("Sockets library successfully opened",0,0,0);
```

Where:

- socket_nethandle is the network handle returned upon success
- Condition value is *errno

Dependencies

The caller must initialize policy_info_ptr being passed to this function by calling dss_init_net_policy_info().

Description

This function opens up the network library, i.e., it assigns a network handle to identify the application in other DS Sockets API functions and sets the application-defined callback functions and the network policy information. Following are more details of the parameters supplied to dss_open_netlib2() function.

The network callback function, net_cb, is provided by the application to receive notifications of the network-related events. The net_cb is of type dss_net_cb_fcn that provides the following information when the callback is invoked:

- nethandle – Identifier of the application for which the network event occurred
- iface_id – Identifier of the interface for which the network event occurred
- errno – Error number to indicate what type of network event occurred. The possible values of error numbers supplied to the network callback are: DS_ENETNONET, DS_ENETISCONN, DS_NETINPROGRESS and DS_ENETCLOSEINPROGRESS.
- net_cb_user_data – Network callback user data supplied at the time of calling dss_open_netlib2().

The socket callback function, sock_cb, is a callback function provided by the application to receive notifications of the socket-related events. sock_cb is of type dss_sock_cb_fcn that provides the following information when the callback is invoked:

- nethandle – Identifier of the application for which the socket event occurred
- sockfd – Identifier of the socket for which the socket event occurred

- event_mask – Event mask indicating which events occurred for the socket. An event mask is formed by ORing the following possible events: DSS_WRITE_EVENT, DSS_READ_EVENT, DSS_CLOSE_EVENT, and DSS_ACCEPT_EVENT

When sock_cb is invoked, the events specified in event_mask are cleared internally and sock_cb is not called again for these events until the caller reselects these events by calling dss_async_select().

- sock_cb_user_data – Socket callback user data supplied at the time of calling dss_open_netlib2()

policy_info_ptr is a pointer to policy information of type dss_net_policy_info_type. It contains the application's network policy to be applied for selection of the network interface to be brought up when dsnet_start() is called. The policy information essentially provides a mechanism to the applications to select the set of network interfaces and to specify generic and specific properties associated with them.

3.1.2 dsnet_get_handle()

Opens the network library and sets the application's network and socket callback functions. It also sets the application's network policy, used to select the network interface to bring-up when dsnet_start() is called.

Parameters

```
sint15 dsnet_get_handle (
                              dss_net_cb_fcn             net_cb,
                         void                       net_cb_user_data,
                              dss_sock_cb_fcn            sock_cb,
                              void                       *sock_)cb_user_data,
                              dss_net_policy_info_type   *policy_info_ptr,
                              sint15                     *errno
                         )
```

→	dsnet_get_handle	

	→ net_cb	Pointer to the network callback function
		This function is to be invoked whenever network related events occur.
		net_cb is of type dss_net_cb_fcn, defined as follows:
		
		`typedef void (*dss_net_cb_fcn)`
		`(`
		`  sint15 nethandle,`
		`  dss_iface_id_type iface_id,`
		`  sint15 errno,`
		`  void *net_cb_user_data`
		`);`

	→ net_cb_user_data	User supplied data to be passed to net_cb whenever invoked

	→	`Sock_cb`	Pointer to the socket callback function
			This function is to be invoked whenever socket related events occur.
			sock_cb is of type dss_sock_cb_fcn, and is defined as follows:
			
			`typedef void (*dss_sock_cb_fcn)`
			`(`
			`  sint15  nethandle,`
			`  sint15  sockfd,`
			`  uint32  event_mask,`
			`  void*  sock_cb_user_data`
			);

	→	`sock_cb_user_data`	User supplied data to be passed to sock_cb whenever invoked

	→	`policy_info_ptr`	Pointer to policy information
			This function uses the following structure to define the network policy to be used to bring up the network when dsnet_start() is called:
			
			`struct dss_net_policy_info_type`
			`{`
			`dss_iface_policy_flags_enum_type policy_flag;`
			`dss_iface_type iface;`
			`boolean ipsec_disabled;`
			`boolean is_routeable;`
			`int family;`
			`struct`
			`{`
			`int pdp_profile_num;`
			`} umts;`
			`};`
			
			The policy_flag field in the dss_net_policy_info_type structure specifies properties of the ifaces to be considered for bring-up and is declared as follows:
			
			`enum dss_iface_policy_flags_enum_type`
			`{`
			`DSS_IFACE_POLICY_SPECIFIED, /* Specified iface */`
			`DSS_IFACE_POLICY_UP_ONLY, /* Only up ifaces */`
			`DSS_IFACE_POLICY_UP_SPEC /* Up ifaces. Use */`
			`/* specified if none is up */`
			`};`
			
			The iface field in dss_net_policy_info_type structure specifies a specific interface or a set of interfaces to be considered for bring-up. One can specify either an iface_id of type dss_iface_id_type (kind field set to DSS_IFACE_ID) or an iface name (kind field set to DSS_IFACE_NAME) of type dss_iface_name_enum_type.
			
			`struct dss_iface_type`
			`{`
			`dss_iface_id_kind_enum_type kind;`
			`union`
			`{`
			`dss_iface_id_type id;`
			`dss_iface_name_enum_type name;`
			`} info;`
			`};`

1

			The following enum declares the validity of dss_iface_name_enum_type:
			
			`enum dss_iface_name_enum_type`
			`{`
			`  DSS_IFACE_CDMA_SN, /* CDMA service network */`
			`  DSS_IFACE_CDMA_AN, /* CDMA authentication n/w */`
			`  DSS_IFACE_UMTS,  /* UMTS network */`
			`  DSS_IFACE_SIO,  /* Serial Interface */`
			`  DSS_IFACE_LO,  /* Local loop-back */`
			`  DSS_IFACE_WWAN,  /* Any wireless WAN */`
			`  DSS_IFACE_ANY_DEFAULT, /* Any default iface */`
			`  DSS_IFACE_RM,  /* Rm iface (unsupported) */`
			`  DSS_IFACE_ANY,  /* Any iface */`
			`  DSS_IFACE_3GPP_ANY, /* Any 3GPP network access Iface. */`
			`  DSS_IFACE_3GPP2_ANY, /* Any 3GPP2 network access Iface. */`
			`  DSS_IFACE_IWLAN_3GPP, /* 3GPP n/w access using WLAN air interface */`
			`  DSS_IFACE_IWLAN_3GPP2 /* 3GPP2 n/w access using WLAN air interface */`
			`};`
			
			The ipsec_disabled field in dss_net_policy_info_type structure specifies whether IPSec protocol should be used over the interface to be brought up.
			The is_routeable field in dss_net_policy_info_type structure should be set to TRUE if the application wants to get a handle on a routable interface. An interface is routable if the interface is brought up by a laptop call. By default, this field is set to FALSE. Only applications that need to modify the behavior of a laptop call should set this flag.
			The family field in dss_net_policy_info_type structure specifies the IP address family for the interface to be brought up. The valid values for this field are as follows:
			`AF_INET  /* IP version 4 */`
			`AF_INET6 /* IP version 6 */`
			`AF_UNSPEC /* Any IP address family */`
			The umts field in dss_net_policy_info_type structure contains the Universal Mobile Telecommunications System (UMTS) specific policy information. Currently, umts structure contains the PDP profile number to be used if a UMTS interface is brought up. The information contained in a umts structure is ignored if the interface brought up is non-UMTS.

	←	`errno`	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EMAPP – No more applications available; maximum number of open applications exceeded ▪ DS_EFAULT – policy_info_ptr is pointing to invalid policy information

Return value

This function returns:

- Network handle on success
- DSS_ERROR on error, and places the error condition value in *errno

Dependencies

The caller must initialize the policy_info_ptr being passed to this function by calling dss_init_net_policy_info().

Description

This function opens up the network library, i.e., it assigns a network handle to identify the application in other DS Sockets API functions and sets the application defined callback functions and the network policy information. Following are more details of the parameters supplied to the dsnet_get_handle() function.

The network callback function, net_cb, is a callback function provided by the application to receive notifications of the network related events. net_cb is of type dss_net_cb_fcn that provides the following information when the callback is invoked:

- nethandle – Identifier of the application for which the network event occurred
- iface_id – Identifier of the interface for which the network event occurred
- errno – Error number to indicate what type of network event occurred. The possible values of error numbers supplied to the network callback are:
  - DS_ENETNONET
  - DS_ENETISCONN
  - DS_NETINPROGRESS
  - DS_ENETCLOSEINPROGRESS
- net_cb_user_data – The network callback user data supplied at the time of calling dsnet_get_handle()
- The socket callback function, sock_cb, is a callback function provided by the application to receive notifications of the socket related events. sock_cb is of type dss_sock_cb_fcn that provides the following information when the callback is invoked:
- nethandle – Identifier of the application for which the socket event occurred
- sockfd – Identifier of the socket for which the socket event occurred
- event_mask – An event mask indicating which events occurred for the socket. An event mask is formed by ORing the following possible events:
  - DSS_WRITE_EVENT
  - DSS_READ_EVENT
  - DSS_CLOSE_EVENT
  - DSS_ACCEPT_EVENT

When sock_cb is invoked, the events specified in event_mask are cleared internally and sock_cb is not called again for these events until the caller reselects these events by calling dss_async_select().

- sock_cb_user_data – The socket callback user data supplied at the time of calling dsnet_get_handle()
- policy_info_ptr – A pointer to policy information of type dss_net_policy_info_type, containing the application's network policy to be applied for selection of the network interface to be brought up when dsnet_start() is called. The policy information essentially provides a mechanism to the applications to select the set of network interfaces and to specify generic and specific properties associated with them.
- policy_flag – Specifies selection criterion for the interfaces specified by iface field:
  - DSS_IFACE_POLICY_SPECIFIED – Indicates that all the interface(s) specified by iface field qualify for selection
  - DSS_IFACE_POLICY_UP_ONLY – Indicates that only those interfaces, given by iface, qualify that are up. If none of the interfaces specified by iface is up then an attempt to bring up the network will fail.
  - DSS_IFACE_POLICY_UP_SPEC – Indicates that the up interfaces specified by iface should be preferred. However, if none of the interfaces specified by iface is up then a down interface from the set of iface qualifies.
- iface – Specifies a set of network interfaces to be considered for bring-up. This set may have a cardinality of one or more, e.g., DSS_IFACE_CDMA_SN specifies a particular interface while DSS_IFACE_ANY includes all the interfaces supported by DMSS. The DS Sockets API allows the applications to specify the interfaces in two possible manners:
  - An opaque interface handle of type dss_iface_id_type can be obtained by supplying:
  - Interface name, interface instance, and optionally physical link instance
  - Socket descriptor
  - Network handle to dss_iface_get_id()function
  - An interface name of type dss_iface_name_enum_type can specify an interface name to consider all the instances of the specified interface.
    - Interface name DSS_IFACE_3GPP_ANY refers to a set of interfaces that can provide access to the 3GPP backend network. Based on the carrier settings, this access can be over WLAN or over the normal 3GPP interface. Similarly, DSS_IFACE_3GPP2_ANY refers to a set of interfaces that can provide access to the 3GPP2 backend network, which can be over WLAN or over conventional 3GPP2 interface.
    - Interface name DSS_IFACE_IWLAN_3GPP refers to accessing the 3GPP backend network using the WLAN air interface. Similarly DSS_IFACE_IWLAN_3GPP2 refers to accessing the 3GPP2 backend network using the WLAN air interface. These interfaces are based on the Wireless Interworking standards for 3GPP and 3GPP2 respectively.
- ipsec_disabled – A flag that dictates whether IPSec protocol should be disabled over the selected network interface
- family – Indicates the IP address family

- umts – Structure containing information specific to UMTS network interface. (Currently, it contains the PDP profile number to be used for establishing the data call. It is considered an error to have any conflict in the information given by the PDP profile and the policy information. The information in umts structure is ignored if the selected interface is not a UMTS interface.

All the fields in the network policy structure together determine the network interface(s) that qualify for bring-up when dsnet_start() is called. If more than one network interface qualifies based on the supplied policy information, then a specific interface is selected based on internal criteria.

It is mandatory for an application to initialize *policy_info_ptr, passed to dsnet_get_handle() function, by calling dss_init_net_policy_info(). The application can modify various fields of *policy_info_ptr after such initialization.

An application not interested in a specific network policy may also supply NULL as the policy_info_ptr, in which case the default network policy is applied.

NOTE: The success of the dsnet_get_handle() call does not necessarily indicate the success of a subsequent dsnet_start() call, i.e., an application may successfully specify a network policy to dsnet_get_handle(), indicating that only UP interfaces should be considered and that only a DSS_IFACE_CDMA_SN interface should be considered. However, if DSS_IFACE_CDMA_SN is not up when the dsnet_start() function is called, the latter will fail with the appropriate failure status. errno is a pointer to the return error value. The dsnet_get_handle() function is a synchronous call and it is called from the context of the socket application task.

Example

```
void my_net_cb
(
sint15            nethandle,
dss_iface_id_type iface_fd,
sint15            errno,
void*             user_data
)
{
/*-------------------------------------------------------------------
  store the callback related information in a buffer, put it on a
  queue and signal my application task to process the network event.
-------------------------------------------------------------------*/
net_ev_buf.nethandle  = nethandle;
net_ev_buf.iface_id   = iface_id;
net_ev_buf.errno      = errno;
net_ev_buf.user_data  = user_data;

q_put(&my_net_ev_q, &(net_ev_buf.link))
  (void)rex_set_sigs(&my_tcb, MY_NET_EV_SIG);
}
```

```
void my_sock_cb
(
sint15 nethandle,
sint15 sockfd,
uint32 event_mask,
void*  user_data
)
{
/*-----------------------------------------------------------------
  store the callback related information in a buffer, put it on a
  queue and signal my application task to process the socket event.
-----------------------------------------------------------------*/
sock_ev_buf.nethandle = nethandle;
sock_ev_buf.sockfd = sockfd;
sock_ev_buf.event_mask = event_mask;
sock_ev_buf.user_data = user_data;

q_put(&my_sock_ev_q, &(sock_ev_buf.link))
  (void)rex_set_sigs(&my_tcb, MY_SOCK_EV_SIG);
}
sint15 my_init_net_func
(
  sint15* errno
)
{
  sint15 nethandle;
dss_net_policy_info_type my_net_policy;

  /*-----------------------------------------------------------------
    Initialize the network library, i.e., set the network and socket
    callback functions and specify the network policy. We choose the
    default network policy by simply initializing my_net_policy.
  -----------------------------------------------------------------*/
  dss_init_net_policy_info(&my_net_policy);
  nethandle = dsnet_get_handle(&my_net_cb,
                               NULL,
                               &my_sock_cb,
                               NULL,
                               &my_net_policy,
                               errno);
  if (nethandle == DSS_ERROR)
  {
    MSG_ERROR("Could not open network library, error=%d", *errno,0,0);
}
```

```
        else
          {
          MSG_MED("Successfully opened network library, nethandle=%d",
        nethandle,0,0);
        }

        return nethandle;
        }
```

3.1.3 dss_socket()

This function opens the socket and sets up all of the socket-related data structures. This socket is bound to the network handle and uses the socket callback and network policy associated with the network handle.

Parameters

```
sint15 dss_socket (
                    sint15   nethandle,
                    byte     family,
                    byte     type,
                    byte     protocol,
                    sint15   *errno
                    )
```

→	dss_socket		
	→	nethandle	Network handle
	→	family	Protocol family to be used with socket. It supports only Internet-style addressing (AF_INET/AF_INET6)
	→	type	Type of socket. It supports: ▪ SOCK_STREAM – TCP sockets ▪ SOCK_DGRAM – UDP sockets
	→	protocol	Type of protocol. It supports the following protocols: ▪ IPPROTO_TCP – SOCK_STREAM type of sockets ▪ IPPROTO_UDP – SOCK_DGRAM type of sockets
	←	errno	Pointer to the returned error value passed by reference. Its valid values are: ▪ DS_EAFNOSUPPORT – Address family not supported ▪ DS_EBADAPP – Invalid network handle ▪ DS_EPROTONOSUPPORT – Invalid or unsupported protocol ▪ DS_ESOCKNOSUPPORT – Invalid or unsupported socket parameter, i.e., type ▪ DS_EMFILE – No more sockets available for opening

Return value

This function will return:

- Socket file descriptor on successful creation of the socket
- DSS_ERROR on error, and places the error condition value in *errno

Dependencies

This function is dependent on dss_open_netlib().

Description

The dss_socket() function is essentially a blocking call, i.e., there are no events associated with, or signals set, when a socket is opened. Successful return from this function indicates that the socket is allocated and available for use.

Note that this function must be called to obtain a valid socket descriptor. It is for use with all other socket-related functions. Before any socket functions can be used, i.e., INPUT/OUTPUT, asynchronous notification, etc., this call must have been successfully returned as a valid socket descriptor. The application must also have made a call to dss_open_netlib() to obtain a valid application ID.

Example

```
sock_fd = dss_socket(
   nethandle,
   AF_INET,
   SOCK_STREAM,
   IPPROTO_TCP,
   &error_num);

if (sock_fd == DSS_ERROR)

{
   ERR("Could not open TCP socket",0,0,0);
   return (DSS_ERROR);
   }
```

Where:

- sock_fd = The socket descriptor obtained by the call to dss_socket()
- nethandle = Application ID obtained by opening the network library with dss_open_netlib()

3.1.4 dss_socket2()

This function opens the socket and sets up all of the socket-related data structures. This socket is not bound to any network handle and uses the user-specified network policy and socket callback.

```
sint15 dss_socket2 (
                      byte                      family,
                      byte                      type,
                      byte                      protocol,
                      sint15                    *errno
                      dss_sock_cb_fcn           sock_cb,
                      void                      *sock_cb_user_data
                      dss_net_policy_info_type  *policy_info_ptr,
                      sint15                    *errno
                      }
```

→	dss_socket2		
	→	family	Protocol family to be used with socket The mobile supports only Internet-style addressing (AF_INET/AF_INET6)
	→	type	Type of socket The mobile supports: ▪ SOCK_STREAM – TCP sockets ▪ SOCK_DGRAM – UDP sockets
	→	protocol	Type of protocol This API supports the following protocols: ▪ IPPROTO_TCP – For SOCK_STREAM type of sockets ▪ IPPROTO_UDP – For SOCK_DGRAM type of sockets

	→	Sock_cb	Pointer to the socket callback function
			This function is invoked to inform the application whenever socket related events of interest occur. sock_cb is of type dss_sock_cb_fcn, defined as follows:
			
			`typedef void (*dss_sock_cb_fcn)`
			`(`
			`  sint15 nethandle,`
			`  sint15 sockfd,`
			`  uint32 event_mask,`
			`  void* sock_cb_user_data`
			`);`

	→	sock_cb_user_data	User supplied data to be passed to sock_cb whenever invoked
	→	policy_info_ptr	Pointer to the policy information structure

	←	errno	Pointer to the returned error value passed by reference; valid values are: ▪ DS_EAFNOSUPPORT – Address family not supported ▪ DS_EBADAPP – Invalid application ID ▪ DS_EPROTONOSUPPORT – Invalid or unsupported protocol ▪ DS_ESOCKNOSUPPORT – Invalid or unsupported socket parameter, i.e., type ▪ DS_EMFILE – No more sockets available for opening

Return value

This function returns:

- Socket file descriptor on successful creation of the socket
- DSS_ERROR on error, and places the error condition value in *errno

Dependencies

The caller must initialize the policy_info_ptr being passed to this function by calling dss_init_net_policy_info().

Description

The dss_socket2() function is essentially a blocking call, i.e., there are no events associated with, or signals set, when a socket is opened. Successful return from this function indicates that the socket is allocated and available for use.

Note that this function must be called to obtain a valid socket descriptor. It is for use with all other socket-related functions. Before any socket functions can be used, i.e., INPUT/OUTPUT, asynchronous notification, etc., this call must have been successfully returned as a valid socket descriptor.

Example

```
       sint15 sock_fd;
     dss_net_policy_info_type my_net_policy;

/*-------------------------------------------------------------------
  We choose the default network policy by simply initializing
  my_net_policy.
-------------------------------------------------------------------*/
 dss_init_net_policy_info(&my_net_policy);

sock_fd = dss_socket2(
   AF_INET,
   SOCK_STREAM,
   IPPROTO_TCP,
   &my_sock_cb,
   NULL,
   &my_net_policy
   &error_num);
```

```
if (sock_fd == DSS_ERROR)

{
   ERR("Could not open TCP socket",0,0,0);
   return (DSS_ERROR);
   }
```

Where:

- sock_fd = The socket descriptor obtained by the call to dss_socket()
- my_sock_cb = Socket callback function

3.1.5 dss_bind()

This function is for all client sockets. It fills the local address and port values in the socket control block.

```
sint15 dss_bind (
                    sint15      sockfd,
                    struct      sockaddr      *localaddr,
                    uint16      addrlen,
                    sint15      *errno
                    }
```

→	dss_bind	
	→ sockfd	Socket descriptor

	→ localaddr	Source IP address and port number
		Related address data structures are defined as follows:
		`struct sockaddr`
		`{`
		`  uint8  sa_family;`
		`  char  sa_data[14];`
		`};`
		`struct in_addr`
		`{`
		`  uint 32 s_addr;`
		`}`
		`struct in6_addr`
		`{`
		`  union`
		`  {`
		`  uint8  u6_addr8[16];`
		`  uint16  u6_addr16[8];`
		`  uint32  u6_addr32[4];`
		`  uint64  u6_addr64[2];`
		`  }in6_u;`
		`};`
		`struct  sockaddr_in`
		`{`
		`  uint8  sin_family;`
		`  uint16  sin_port;`
		`  struct  in_addr  sin_addr;`
		`  char  sin_zero[8];`
		`};`
		`struct  sockaddr_in6`

		`}`
		`  uint8  sin_family;`
		`  uint16  sin_port;`
		`  uint32  sin6_flowinfo;`
		`  struct  in6_addr  sin6_addr;`
		`  uint32  sin6_scope_id;`
		`};`

	→ addrlen	Length of localaddr It should be the size of (struct sockaddr)

	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EEBADF – Invalid socket descriptor was specified ▪ DS_EAFNOSUPPORT – Address family not supported ▪ DS_EOPNOTSUPP – Operation not supported, possibly because a specific IP address binding was requested ▪ DS_EADDRINUSE – Specified address is already in use by another socket ▪ DS_EINVAL – Socket is already attached to a local name ▪ DS_EFAULT – Invalid address and/or invalid address length

Return value

This function returns:

- DSS_SUCCESS on successful operation
- DSS_ERROR on error, and places the error condition value in *errno

Description

This call attaches a local address and port number to the specified socket. This function must receive, as a parameter, a valid socket descriptor obtained from a previous successful call to dss_socket().

If the call is not explicitly issued, a client socket implicitly binds in calls to dss_connect() or dss_sendto(). This function supports binding to multicast addresses for use with multicast-multicast interfaces such as Broadcast Multicast Services (BCMCS). This function also supports binding to local IPv6 addresses. This function does not, however, support binding a local IPv4 address and should only bind to a local port number, i.e., dss_bind() should always be called with IPADDR_ANY as the IP address. The sockets library automatically assigns the local IP address if one is not specified. In addition, when this function is called with the port number as 0 in the local address parameter passed to it, the sockets library assigns a free port as well to the socket. The port number assigned in this manner is made available to the caller in the value result parameter localaddr.sin_port.

When applications wish to use a multicast interface such as BCMCS or MediaFLO™, they must also explicitly bind the socket using dss_bind() to the multicast IP and port submitted in the multicast join call. If the application does not explicitly bind to this IP and port, it will not receive the multicast data on the socket. For more information on joining a multicast group, refer to the multicast join IOCTL located in Section 0.

If applications wish to unbind to their IP address and port, dss_bind() may be called with the localaddr pointer passed as NULL. This will cause the socket to reset the IP and port binding on the socket to zero. For IPv6 privacy addresses, this action is required before binding to a new privacy address. See Section 8.1 for more information on IPv6 privacy extensions.

Even though dss_bind() can be called with IPADDR_ANY or IN6ADDR_ANY, the bind call restricts the port binding to the interfaces satisfying the network policy of the socket. In the case of sockets created using the dss_socket() call, the data transfer is restricted to one interface that is brought up with a dsnet_start() call by the application owning the socket. In the case of sockets created using the dss_socket2() call, the data transfer is restricted to all the up interfaces satisfying the network policy of the socket. The only exception is the loopback interface. All sockets can send and receive data on a loopback interface, regardless of their network policy.

Even though we bind a socket to a set of interfaces based on network policy, the port space for sockets is unique. Two sockets cannot bind to the same port even if their network policy points to the different interfaces.

Example

```
/*-------------------------------------------------------------------
  Socket successfully opened. Fill in localaddr struct.
  Remember to use network byte ordering.
-------------------------------------------------------------------*/
 memset((char *) &localaddr, 0, sizeof(localaddr));
 localaddr.sin_family      = AF_INET;
 localaddr.sin_addr.s_addr = INADDR_ANY;
 localaddr.sin_port        = dss_htons(browser_client_port_num);

/*-------------------------------------------------------------------
  Now bind the local port number to the socket.
-------------------------------------------------------------------*/

MSG_LOW("Binding Socket",0,0,0);

ret_val = dss_bind( sock_fd,
                        (struct sockaddr *)&localaddr,
                         sizeof(localaddr),
                         &error_num
                        );
```

3.2 Client

3.2.1 dss_connect()

For TCP sockets, this call attempts to establish the TCP connection. For UDP sockets, this call associates the socket descriptor with the specified server address and, subsequently, the socket can send datagrams to or receive datagrams only from this server address.

Parameters

```
sint15 dss_connect (
                      sint15      sockfd,
                      struct      sockaddr  *servaddr,
                      uint16      addrlen,
                      sint15      *errno
                      )
```

→	dss_connect	
	→ sockfd	Socket descriptor

	→ sockaddr	struct sockaddr
		{
		uint8 sa_family;
		char sa_data[14];
		};
		struct in_addr
		{
		uint 32 s_addr;
		}
		struct in6_addr
		{
		union
		{
		uint8 u6_addr8[16];
		uint16 u6_addr16[8];
		uint32 u6_addr32[4];
		uint64 u6_addr64[2];
		}in6_u;
		};
		struct sockaddr_in
		{
		uint8 sin_family;
		uint16 sin_port;
		struct in_addr sin_addr;
		char sin_zero[8];
		};
		struct sockaddr_in6

			}
			  uint8 sin_family;
			  uint16 sin_port;
			  struct in_addr sin_addr;
			  char sin_zero[8];
			};
			struct sockaddr_in6
			}
			  uint8 sin_family;
			  uint16 sin_port;
			  uint32 sin6_flowinfo;
			  struct in6_addr sin6_addr;
			  uint32 sin6_scope_id;
			};

	→	addrlen	Length of servaddr It should be the size of (struct sockaddr)
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EWOULDBLOCK – Operation will block ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_ECONNREFUSED – Connection attempt was refused due to the receipt of a server RST ▪ DS_ETIMEDOUT – Connection establishment timed out ▪ DS_EFAULT – Invalid addrlen (address length) parameter ▪ DS_EIPADDRCHANGED – As part of IP gateway handoff, the network level IP address changed ▪ DS_EINPROGRESS – Connection establishment in progress ▪ DS_EISCONN – Specified socket descriptor is already connected ▪ DS_ENETDOWN – Underlying network subsystem is unavailable ▪ DS_EOPNOTSUPP – Invalid servaddr (server address) specified ▪ DS_EINVAL – Socket is already connected but to an address different from the one specified ▪ DS_ENOROUTE – No route to destination found ▪ DS_EADDRREQ – Destination address required In addition to the above error codes, the following error codes can be returned for a stream socket in case the DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- DSS_ERROR and places the error condition value in *errno

Dependencies

The network subsystem must be established and available.

Description

The underlying network subsystem must be available, i.e., a route to the destination must be present for dss_connect() to succeed. dss_connect() determines the outgoing network interface based on the server address and in case this interface is not up, it returns DS_ENETDOWN. For traffic destined outside the mobile station, an application has to call ppp_open() to bring up the network interface before attempting to connect a socket. For traffic between applications on the mobile station, dss_connect() can be used directly because the loopback interface is always up.

For TCP sockets, if the network is available, dss_connect() returns DS_EWOULDBLOCK. The client should call dss_async_select(DS_WRITE_EVENT). When the socket is connected (making the socket writable), or the network library gives up on trying to connect (TCP connection refused, timed-out, etc.), DS_WRITE_EVENT will be asserted. The application callback function is invoked to provide event notification, and the application can call dss_getnextevent() to determine which event occurred. If DS_WRITE_EVENT has been asserted, the application should make calls to dss_connect() until TCP has been established, indicated by a return value of DSS_ERROR with the error condition set to DS_EISCONN.

For UDP sockets, the connect call associates the socket descriptor with the specified server address. After dss_connect() is called, a UDP socket can send datagrams to and receive datagrams from only the address it is connected to. An application can change the association of a UDP socket by connecting multiple times and can dissolve an association by connecting to a NULL address. Note that while dss_connect() does not result in a communication with the peer for a UDP socket, it does determine the outgoing network interface based on the server address and verifies that this interface is up. In case the network interface is not up, the caller gets a DS_ENETDOWN error as in the case of dss_connect() for TCP sockets.

Example

Consider a connection attempt to www.qualcomm.com, which is mapped to the class C IP address 192.35.156.18 and uses well-known port 80. The hexadecimal representation of this IP address is:

```
Server_ip = 0xC0239C12
```

and

```
port_num = 80
```

The function call can then be used as:

```
/*-------------------------------------------------------------------
  Socket successfully opened. Fill in localaddr struct.
  Remember to use network byte-ordering.
-------------------------------------------------------------------*/
 memset((char *) &serv_addr, 0, sizeof(serv_addr));
 serv_addr.sin_family      = AF_INET;
 serv_addr.sin_addr.s_addr = dss_htonl(server_ip);
 serv_addr.sin_port        = dss_htons(port_num);

/*-------------------------------------------------------------------
  Try to connect to the destination server. This is a
  socket call; we need to specify via async_select the
  signals we are interested in. In this case, we only
  need to select DS_WRITE_EVENT. That event occurs if
  either the connection is made or if it fails.
-------------------------------------------------------------------*/

MSG_MED("Connecting to destination server",0,0,0);
do
      {
ret_val = dss_connect( sock_fd,
                           (struct sockaddr *)&serv_addr,
                            sizeof(serv_addr),
                            &error_num
                            );
if ((ret_val == DSS_SUCCESS) ||
         ((ret_val == DSS_ERROR) && (error_num == DS_EISCONN))
        )
     {
       MSG_HIGH("    Socket connected without waiting",0,0,0);
       Break;
     }
     else if ((ret_val == DSS_ERROR) &&
              (error_num == DS_EWOULDBLOCK))
     {
```

```
                /* Since we have done the async_select above, we can now
                 * wait on the socket callback signal. After the signal
                 * returns, we will know that something has finally
                 * happened with that socket.
                 */
                 MSG_HIGH("     EwouldBlock on connect...",0,0,0);
                 Rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
                 Dss_async_select(sock_fd,DS_WRITE_EVENT, &error_num);
                 app_wait(APP_SOCKET_CB_SIG);
            }
            else
            {
               /* We got some other error... since we are just
                * starting out, the only kind of benign error would
                * be DS_EINPROGRESS or DS_EISCONN. However, these
                * should not be returned, as we are the only application
                * doing sockets work*/
               ERR("Could not connect to server",0,0,0);
               Return ERROR;
            }
         }
         while (1);
         return sock_fd;
```

3.3 Server

3.3.1 dss_listen()

This function initiates passive open for TCP connections.

Parameters

```
sint15 dss_listen
                    (
                    sint15    sockfd,
                    sint15    backlog,
                    sint15    *dss_errno
                    )
```

→	dss_listen		
	→	sockfd	Socket descriptor
	→	backlog	Maximum number of half-open TCP connections to track at one time
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_EOPNOTSUPP – Socket is not capable of listening (UDP) ▪ DS_EFAULT – Backlog parameter invalid ▪ DS_NOMEM – Not enough memory to establish backlog connections ▪ DS_EINVAL – Socket already open, closed, unbound, or not one you can listen on

Return value

This function returns:

- On error – DSS_ERROR and places the error condition value in *errno
- On success – DSS_SUCCESS

Dependencies

dss_bind must be called on this socket before dss_listen.

Description

For TCP, this starts a passive open for connections. Upon a successful connection, the socket callback function is invoked asserting DS_ACCEPT_EVENT as TRUE. The application should respond with a call to dss_accept(). If a connection is received and there are no free queue slots, the new connection is rejected (ECONNREFUSED). The backlog queue is for all unaccepted sockets (half-open, or completely established).

Once an application calls dss_listen on a socket, it is allowed to accept connection requests on all the open interfaces. However, no guarantees are given for connections accepted on the interfaces not bound to the application as these interfaces can go down at any time, thus closing the connection.

A listening UDP does not make sense, and as such, is not supported. DS_EOPNOTSUPP is returned.

The sockfd parameter is a created (dss_socket) and bound (dss_bind) socket that will become the new listening socket. The backlog parameter indicates the maximum length for the queue of pending sockets. If backlog is larger than the maximum, it will be reduced to the maximum (see DSS_SOMAXCONN). This is the BSD behavior.

NOTE: DSS_SOMAXCONN represents the maximum number of supported listening sockets.

Example

```
ret_val = dss_listen (sockfd, DS_SOMAXCONN, &errno);
```

3.3.2 dss_accept()

This function returns the descriptor for a new passively opened connection.

Parameters

```
sint15 dss_accept
                    (
                    sint15          sockfd,
                    struct  sockaddr *remoteaddr,
                    uint16          *addrlen,
                    sint15          *dss_errno
                    )
```

→	dss_accept		
	→	sockfd	Socket descriptor
	→	remoteaddr	Pointer to a block describing the address of the remote end of the new connection The specific type will depend on the underlying protocol. When remoteaddr is NULL, nothing is filled in
	↔	addrlen	Pointer to the length of the remoteaddr is handed in The length of the actual address structure will be returned

	←	dss_errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_EWOULDBLOCK – Operation would block ▪ DS_EOPNOTSUPP – Socket is not of type SOCK_STREAM ▪ DS_EINVAL – Socket is not listening ▪ DS_EFAULT – sockaddr or addrlen parameter is bogus ▪ DS_ENETNONET – Network subsystem unavailable for some unknown reason ▪ DS_ENETINPROGRESS – Network subsystem establishment currently in progress ▪ DS_ENETCLOSEINPROGRESS – Network subsystem close in progress ▪ DS_NOMEM – Not enough memory to establish backlog connections

Return value

This function returns:

- An integer – Indicating the descriptor of the new socket
- DSS_ERROR – Error; error condition value is placed in *errno

Description

The accept function is used on listening sockets to respond when DS_ACCEPT_EVENT is asserted. The first backlog queued connection is removed from the queue and bound to a new connected socket as if dss_socket was called. The newly created socket is in the connected state. The listening socket is unaffected and the queue size is maintained, i.e., there is no need to call listen again.

Example

```
struct sockaddr_in in;
uint16 len = sizeof(in);
newsockfd = dss_accept( sockfd, &in, &len, &errno);
```

3.4 Event management

3.4.1 dss_async_select()

This function enables the events (for which notification is requested) through the asynchronous notification mechanism.

Parameters

```
sint31 dss_async_select (
                          sint15          sockfd,
                          sint31          interest_mask,
                          sint15          *errno
                              )
```

→	dss_async_select		
	→	sockfd	Socket descriptor
	→	interest_mask	Specifies the bit mask containing the events in which the application is interested and for which the application will receive asynchronous notification
	←	errno	Pointer to the returned error value that is passed by reference; valid value is: ▪ DS_EBADF – Invalid socket descriptor was specified

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in *errno

Description

The application specifies a bit mask of events in which it is interested. Using its application callback function, the application will receive asynchronous notification about the specified events. For a list of the asynchronous socket events, see Table 2-1. dss_async_select() also performs a real-time check to see if any of the events have already occurred. If any of the events have been asserted, the function invokes the application callback function to notify the application that the event has occurred.

Example

```
select_mask = (DS_WRITE_EVENT | DS_CLOSE_EVENT);
ret_val = dss_async_select(sock_fd,select_mask,&error_num);
```

3.4.2 dss_async_deselect()

This function clears events that have been set through dss_async_select().

Parameters

```
sint31 dss_async_deselect (
                          sint15          sockfd,
                          sint31          clr_interest_mask,
                          sint15          *errno
                          )
```

→	dss_async_deselect		
	→	sockfd	Socket descriptor
	→	clr_interest_ mask	Specifies bit mask of the events to be cleared The application will no longer receive notification for these events. If the specified event was not set, the clear request is silently discarded.
	←	errno	Pointer to the returned error value that is passed by reference; valid value is: ▪ DS_EBADF – Invalid socket descriptor was specified

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in *errno

Description

As part of the asynchronous notification mechanism, the application needs a provision for clearing events that have been set through dss_async_select(). The application specifies a bit mask of events that it wishes to clear – events for which it will no longer receive notification. If the bit is already cleared in the bit mask, the request is silently discarded.

Example

```
deselect_mask = (DS_WRITE_EVENT | DS_CLOSE_EVENT);
ret_val = dss_async_deselect(sock_fd,deselect_mask,&error_num);
```

3.4.3 dss_getnextevent()

This function performs a real-time check to determine which events have occurred and clears the corresponding bits in the interest mask.

Parameters

```
sint31 dss_getnextevent (
                              sint15      nethandle,
                              sint15      *sockfd_ptr,
                              sint15      *errno
                              )
```

→	dss_getnextevent		
	→	nethandle	Application ID
	→	sockfd_ptr	Pointer to socket descriptor
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_EBADAPP – Invalid network handle was specified

Return value

This function returns:

- Event mask of the events that were asserted for the specified socket descriptor.
- 0 – No events have occurred
- DSS_ERROR – Error; error condition value placed in *errno
- On passing *sockfd_ptr with a NULL socket descriptor, places the next available socket descriptor in *sockfd_ptr and returns the event mask for that socket

Description

This function performs a real-time check to determine if any of the events of interest specified in the socket control block's event mask have occurred. It also clears any bits in the event mask that have occurred. The application must re-enable these events through a subsequent call to dss_async_select(). The application may pass in a pointer to a single-socket descriptor to determine if any events have occurred for that socket.

Alternatively, the application may set this pointer's value to NULL (not to be confused with a NULL pointer, but rather a pointer to a socket descriptor whose value is NULL), in which case the function will return values for the next available sockets. The next available socket's descriptor will be placed in the socket descriptor pointer, and the function will return. If no sockets are available (no events have occurred across all sockets for that application), the pointer value will remain NULL (originally the value that was passed in), and the function will return 0, indicating that no events have occurred.

Special considerations

The function clears any bits in the event mask that have occurred. The application must re-enable these events through a subsequent call to dss_async_select(). Also, this function cannot be used in conjunction with dss_opennetlib2() as the event mask is cleared as soon as an event callback is called.

Example

```
sint15  nethandle;
sint31  event_mask;
sint15  error_num;
sint15  sock_fd;

event_mask = dss_getnextevent(nethandle, &sock_fd, &error_num);

if (event_mask & DS_CLOSE_EVENT)
{
   MSG_HIGH("DS_CLOSE_EVENT was asserted.",0,0,0);
}
else if (event_mask & DS_READ_EVENT)
{
   MSG_HIGH("DS_READ_EVENT was asserted.",0,0,0);
}
else if (event_mask & DS_WRITE_EVENT)
{
   MSG_HIGH("DS_WRITE_EVENT was asserted.",0,0,0);
}
else if (event_mask == 0)
{
   MSG_HIGH("No events were asserted.",0,0,0);
}
```

3.5 Input

3.5.1 dss_read()

This function reads a specified number of bytes into a buffer over the TCP/UDP transport.

Parameters

```
sint15 dss_read (
                  sint15   sockfd,
                  byte     *buffer,
                  uint16   nbytes,
                  sint15   *errno
                  )
```

→	dss_read		
	→	sockfd	Socket descriptor
	→	*buffer	Data buffer from which data is copied
	→	nbytes	Specifies the maximum number of bytes to be read
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_ENOTCONN – Nonexistent TCP/UDP connection ▪ DS_ECONNRESET – TCP connection was reset by the server ▪ DS_ECONNABORTED – TCP connection was aborted due to timeout or other failure ▪ DS_EIPADDRCHANGED – As part of the IP gateway handoff, the network-level IP address changed, causing the TCP connection to reset ▪ DS_EPIPE – Peer closed the TCP connection ▪ DS_ENETDOWN – Network subsystem is unavailable ▪ DS_EFAULT – Bad buffer address ▪ DS_EMSGTRUNC – UDP datagram truncated because the supplied buffer is too small ▪ DS_EWOULDBLOCK – No data is available for reading ▪ DS_EEOF – End of file received or a read operation is being attempted after closing down the read-half of the connection In addition to the above error codes, the following error codes can be returned in case there is no data to read and DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- Total number of bytes read, which could be less than the number of bytes specified.

NOTE: For DS_EEOF errors, this return value will be zero.

- On error, it returns DSS_ERROR and places the value in *errno.
- An additional return value exists in the *errno variable. If this value is nonzero, a valid return value exists in this variable and should be handled accordingly.

Description

If there are no bytes available to be read, the call returns DS_EWOULDBLOCK in the errno variable. The application then needs to call dss_async_select() specifying the DS_READ_EVENT. When the DS_READ_EVENT is asserted, the Application Callback function is invoked. The application should then use dss_getnextevent() to determine if DS_READ_EVENT is true. If true, the application should make another call to dss_read().

Example

```
bytes_per_read = dss_read(sock_fd,
                          (byte *)(buffer),
                           BYTES_TO_READ,
                           &error_num);

if ( (bytes_per_read == DSS_ERROR) && (error_num == DS_EWOULDBLOCK) )
{
  rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
  dss_async_select(sock_fd,DS_READ_EVENT,
                   &error_num);
}
```

3.5.2 dss_readv()

This function reads iovcount number of bytes across all buffers from the TCP/UDP transport.

Provides the scatter read variant of the dss_read() call that allows the application to read into noncontiguous buffers.

Parameters

```
Sint15 dss_readv (
                    sint15      sockfd,
                    struct      iovec iov [],
                    sint15      iovcount,
                    sint15      *errno
                    )
```

→	dss_readv		
	→	sockfd	Socket descriptor

	→	iov [ ]	Data buffer to which data is copied
			The buffer is owned by the application. It is the responsibility of the application to allocate enough buffer space for at least iovcount bytes.
			The following buffer data structure is defined:
			`struct iovec {`
			`  byte  *iov_base; /* starting address of buffer*/`
			`  uint16 iov_len;  /* size of the buffer in bytes*/`
			`}`

	→	iovcount	Specifies the total number of bytes to be read (total iovlen across the number of buffers)

	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_ENOTCONN – No existing TCP/UDP connection ▪ DS_ECONNRESET – TCP connection was reset by the server ▪ DS_ECONNABORTED – TCP connection was aborted due to timeout or other failure ▪ DS_EIPADDRCHANGED – As part of IP gateway handoff, the network-level IP address changed, causing the TCP connection to reset ▪ DS_EPIPE – Peer closed the TCP connection ▪ DS_ENETDOWN – Network subsystem is unavailable ▪ DS_EFAULT – Bad buffer address ▪ DS_EMSGTRUNC – UDP datagram truncated because the supplied buffer is too small ▪ DS_EWOULDBLOCK – No data is available for reading ▪ DS_EEOF – End of file received or a read operation is being attempted after closing down the read-half of the connection In addition to the above error codes,the following error codes can be returned in case there is no data to read and the DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- Total number of bytes read, which could be less than the number of bytes specified. Note that for DS_EEOF errors, this return value will be zero.
- On error, it returns DSS_ERROR and places the value in *errno.
- An additional return value exists in the *errno variable. If this value is nonzero, a valid return value exists in this variable and should be handled accordingly.

Description

If there are no bytes available for reading, the call returns DS_EWOULDBLOCK in the errno variable. The application then needs to call dss_async_select() specifying the DS_READ_EVENT. When the DS_READ_EVENT is asserted, the Application Callback function is invoked. The application should then use dss_getnextevent() to determine if DS_READ_EVENT is true. If true, the application should make another call to dss_readv().

Example

```
bytes_per_read = dss_readv(sock_fd,
                           iov,
                           iov_count,
                           &error_num);

if ( (bytes_per_read == DSS_ERROR) && (error_num == DS_EWOULDBLOCK) )
{
  rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
  dss_async_select(sock_fd,DS_READ_EVENT,
                   &error_num);
}
```

3.5.3 dss_recvfrom()

This function reads nbytes bytes into the buffer over the UDP transport.

Parameters

```
sint15 dss_recvfrom (
                      sint15    sockfd,
                      byte      *buffer,
                      uint16    nbytes,
                      uint16    flags,
                      struct    Sockaddr  *fromaddr,
                      uint16    *addrlen,
                      sint15    *errno
                      )
```

→	dss_recvfrom		
	→	sockfd	Socket descriptor
	→	buffer	Data buffer into which data is copied The buffer is owned by the application. It is the responsibility of the application to have enough buffer space allocated for at least nbytes bytes.
	→	nbytes	Maximum number of bytes to be read
	→	flags	Not supported – Must be set to zero

	←	fromaddr	If the fromaddr is non-NULL, then IP address and port number is filled in with the values from the sender of the data.
			Related address data structures are defined as follows:
			`Struct sockaddr`
			`{`
			`  uint8  sa_family;`
			`  char  sa_data[14];`
			`};`
			`struct in_addr`
			`{`
			`  uint 32 s_addr;`
			`}`
			`struct in6_addr`
			`{`
			`  union`
			`  {`
			`  uint8  u6_addr8[16];`
			`  uint16  u6_addr16[8];`
			`  uint32  u6_addr32[4];`
			`  uint64  u6_addr64[2];`
			`  }in6_u;`
			`};`
			`struct  sockaddr_in`
			`{`
			`  uint8  sin_family;`
			`  uint16  sin_port;`
			`  struct  in_addr  sin_addr;`
			`  char  sin_zero[8];`
			`};`
			`struct  sockaddr_in6`
			`{`
			`  uint8  sin_family;`
			`  uint16  sin_port;`
			`  uint32  sin6_flowinfo;`
			`  struct  in6_addr  sin6_addr;`
			`  char  sin6_scope_id;`
			`};`

	↔	addrlen	Pointer to the length of fromaddr **Note:** This value is passed by reference (unlike that in dss_sendto()), and is filled in by this function call.

	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_EAFNOSUPPORT – Address family is not supported ▪ DS_EWOULDBLOCK – No data is available for reading ▪ DS_ENETDOWN – Network subsystem is unavailable ▪ DS_EFAULT – Bad buffer address ▪ DS_EMSGTRUNC – Message truncated because the supplied buffer is too small ▪ DS_EOPNOSUPPORT – Specified option is not supported; this implementation does not support flags In addition to the above error codes, the following error codes can be returned in case there is no data to read and the DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- Actual number of bytes read, which could be less than the number of bytes specified.
- On error, it returns DSS_ERROR and places the value in *errno.
- An additional return value exists in the *errno variable. If this value is nonzero, a valid return value exists in this variable and should be handled accordingly.

Description

This function reads nbytes bytes into the buffer over the UDP transport.

If there are no bytes available for reading, the call returns DS_EWOULDBLOCK in the errno variable. The application then needs to call dss_async_select() specifying the DS_READ_EVENT. When the DS_READ_EVENT is asserted, the application callback function is invoked. The application should then use dss_getnextevent() to determine if DS_READ_EVENT is true. If true, the application should make another call to dss_recvfrom().

Special considerations

This implementation does not support TCP sockets; therefore, any calls to dss_recvfrom() with a TCP socket will return an error.

For sockets created by dss_socket() call, data can be received only on the interface brought up by the application owning the socket. For sockets created by the dss_socket2() call, the data can only be received on a set of interfaces that satisfy the network policy of the socket.

Example

```
/*-----------------------------------------------------
    Socket successfully opened. Clear from_addr struct.
    The structure will be filled in by the call to
    dss_recvfrom(), along with the address length
    parameter.
  -----------------------------------------------------*/
  memset((char *) &from_addr, 0, sizeof(from_addr));
  flags_are_unsupported     = 0;

  /*-----------------------------------------------------
    Now receive datagrams using dss_recvfrom().
  -----------------------------------------------------*/
  MSG_MED("Receiving datagrams from server",0,0,0);

  bytes_per_read = dss_recvfrom( sock_fd,
                                (byte *) buffer,
                                 BYTES_TO_READ,
                                 flags_are_unsupported,
                                  (struct sockaddr *)&from_addr,
                                 &from_length,
                                 &error_num
                               );

  if ((bytes_per_read == DSS_ERROR) && (error_num == DS_EWOULDBLOCK))
  {
    rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
    dss_async_select(sock_fd,DS_READ_EVENT,
                     &error_num);
  }
```

3.5.4 dss_recvmsg()

This function reads data and ancillary data into the msghdr struct over the transport specified by the socket descriptor.

Parameters

```
sint15 dss_recvmsg (
                        sint15    sockfd,
                        struct    dss_msghdr     *msg,
                        uint16    flags,
                        sint15    *errno
                        )
```

→	dss_recvmsg		
	→	Sockfd	Socket descriptor

	→	dss_msghdr	It is the responsibility of the application to allocate and free the memory for dss_msghdr.
			
			`struct dss_msghdr`
			`{`
			`  void  * msg_name;`
			`  uint16  msg_namelen;`
			`  struct iovec  * msg_iov;`
			`  uint16  msg_iovlen;`
			`  void  * msg_control;`
			`  uint16  msg_controllen;`
			`  int  msg_flags;`
			`};`
			
			msg_name is of type struct sockaddr.
			msg_namelen should be the size of the sockaddr being sent.
			msg_iov is of type:
			
			`struct iovec`
			`{`
			`  byte *iov_base;`
			`  uint16 iov_len;`
			`};`
			
			This buffer (iov_base) is owned by the application. It is the responsibility of the application to have enough buffer space allocated for at least iov_len bytes.
			msg_iovlen represents the number of elements of msg_iov.
			msg_control is for ancillary data. It is the responsibility of the application to allocate the memory for any data structures that are to be returned. Cmsghdr struct will be returned through msg_control and will contain the ancillary data.
			
			`struct dss_cmsghdr`
			`{`
			`  uint16  cmsg_len; /* data byte count, including header */`
			`  int16  cmsg_level;  /* originating protocol */`
			`  int16  cmsg_type;  /* protocol-specific type */`
			`}`
			
			msg_controllen is the total size in bytes of the control structure returned. The application should set msg_controllen to the length of the buffer it allocated for receiving ancillary data.
			flags is currently not supported and should be set to 0.

	→	flags	Only DSS_MSG_ERRQUEUE flag is supported which is used for retrieving enqueued ICMP errors in case DSS_IP_RECVERR socket option is enabled. Otherwise it should be set to 0.

	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_EAFNOSUPPORT – Address family is not supported ▪ DS_EWOULDBLOCK – No data is available for reading ▪ DS_ENETDOWN – Network subsystem is unavailable ▪ DS_EFAULT – Bad buffer address ▪ DS_EMSGTRUNC – Message truncated because the supplied buffer is too small ▪ DS_EOPNOSUPPORT – Specified option is not supported; this implementation does not support flags In addition to the above error codes, the following error codes can be returned in case there is no data to read and the DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- Actual number of bytes read which could be less than the number of bytes specified.
- On error, it returns DSS_ERROR and places the value in *errno.
- An additional return value exists in the *errno variable. If this value is nonzero, a valid return value exists in this variable and should be handled accordingly.

Description

This function is a common read function for all the sockets. The message header contains an array of scattered buffers, a socket descriptor, and an address field for filling the source address of the received packet. The function reads data into the scattered buffers over the transport specified by the socket descriptor.

The ancillary data is returned through msg_control in the form of a chain of cmsghdrs. Applications expecting ancillary data should allocate an appropriately large buffer for msg_control and set the msg_controllen to the length of the allocated buffer *before* making the dss_recvmsg call. Applications can parse the returned ancillary data containing cmsghdrs using cmsg macros. The macros to be used in the standard BSD style are DSS_CMSG_DATA, DSS_CMSG_FIRSTHDR, DSS_CMSG_NXTHDR, DSS_CMSG_ALIGN, DSS_CMSG_SPACE, and DSS_CMSG_LEN.

NOTE: Only the ancillary data associated with the DSS_IP_RECVIF, DSS_IP_RECVTTL, DSS_IP_RECVECN and DSS_IP_RECVERR socket options is currently supported. Refer to the details of these socket options in Section 3.8.1.

If there are no bytes available for reading, the call returns DS_EWOULDBLOCK in the errno variable. The application then needs to call dss_async_select() specifying the DS_READ_EVENT. When DS_READ_EVENT is asserted, the application callback function is invoked. The application should then use dss_getnextevent() to determine if DS_READ_EVENT is true. If true, the application should make another call to dss_recvmsg().

For sockets created by the dss_socket() call, data can be received only on the interface brought up by the application owning the socket. For sockets created by the dss_socket2() call, the data can be received only on a set of interfaces that satisfy the network policy of the socket.

Example

```
/*-----------------------------------------------------
  Now receive datagrams using dss_recvmsg().
-----------------------------------------------------*/

bytes_per_read = dss_recvmsg( sock_fd,
                              &msghdr,
                              flags_are_unsupported,
                              &error_num
                            );

if ((bytes_per_read == DSS_ERROR) && (error_num == DS_EWOULDBLOCK))
{
  rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
  dss_async_select(sock_fd,DS_READ_EVENT,
                   &error_num);
}
```

3.6 Output

3.6.1 dss_write()

This function sends a specified number of bytes in the buffer over the TCP/UDP transport.

Parameters

```
sint31 dss_write (
                    sint15  sockfd,
                    const void   *buffer,
                    uint16  nbytes,
                    sint15  *errno
                    )
```

→	dss_write		
	→	sockfd	Socket descriptor
	→	*buffer	Data buffer from which data is copied This buffer is owned by the application. The socket layer does not change the buffer in any way. The application is responsible for freeing the buffer.
	→	nbytes	Number of bytes to be written to the socket

	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_ENOTCONN – No existing TCP/UDP connection ▪ DS_ECONNRESET – TCP connection was reset by the server ▪ DS_ECONNABORTED – TCP connection was aborted due to timeout or other failure ▪ DS_EIPADDRCHANGED – As part of the IP gateway handoff, the network-level IP address changed, causing the TCP connection to reset ▪ DS_EPIPE – Peer closed the TCP connection ▪ DS_ESHUTDOWN – Write operation is being attempted on a socket with the write-half of the connection closed ▪ DS_ENETDOWN – Network subsystem is unavailable ▪ DS_EFAULT – Bad buffer address ▪ DS_EMSGSIZE – Message is too large to be sent all at once over the requested transport ▪ DS_EWOULDBLOCK – No free buffers are available for writing In addition to the above error codes, the following error codes can be returned in case the DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- Total number of bytes written, which could be less than the number of bytes specified.
- On error, it returns DSS_ERROR and places the value in *errno.
- An additional return value exists in the *errno variable. If this value is nonzero, a return value exists in this variable and should be handled accordingly.

Description

Sends nbytes number of bytes in the buffer over the TCP/UDP transport.

If the specified number of bytes to write is not equal to the number of bytes actually written, the call will return the number of bytes written. The application then needs to call dss_async_select() specifying DS_WRITE_EVENT. When the DS_WRITE_EVENT is asserted, the application callback function is invoked. The application should then use dss_getnextevent() to determine if DS_WRITE_EVENT is true. If it is true, the application can make another call to dss_write() to send the remaining bytes.

Example

```
bytes_per_write = dss_write(sock_fd,
                            (byte *)(buffer),
                            BYTES_TO_WRITE,
                            &error_num);

if ((bytes_per_write == DSS_ERROR) && (error_num == DS_EWOULDBLOCK))
{
  rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
  dss_async_select(sock_fd,DS_WRITE_EVENT,
                   &error_num);
}
```

3.6.2 dss_writev()

This function sends an iovcount number of bytes across all buffers over the TCP/UDP transport.

Provides the gather write variant of the dss_write() call that allows the application to write from noncontiguous buffers.

```
Parameterssint15 dss_writev (
                                sint15    sockfd,
                                struct    iovec iov [],
                                sint15    iovcount,
                                sint15    *errno
                                )
```

→	dss_writev		
	→	sockfd	Socket descriptor

	→	iov [ ]	Array of data buffers from which data is copied
			These buffers are owned by the application. The socket layer does not change these buffers in any way. The application is responsible for freeing the array of buffers. The following buffer data structure is defined:
			`Struct iovec {`
			`  Byte  *iov_base;  /* Starting address of buffer  */`
			`  uint16  iov_len;  /* Size of the buffer in bytes */`
			`}`

	→	iovcount	Total number of bytes to be written to the socket (total iovlen across buffers)

	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_ENOTCONN – No existing TCP/UDP connection ▪ DS_ECONNRESET – TCP connection was reset by the server ▪ DS_ECONNABORTED – TCP connection was aborted due to timeout or other failure ▪ DS_EIPADDRCHANGED – As part of the IP gateway handoff, the network-level IP address changed, causing the TCP connection to reset ▪ DS_EPIPE – Peer closed the TCP connection ▪ DS_ESHUTDOWN – Write operation is being attempted on a socket with the write-half of the connection closed ▪ DS_ENETDOWN – Network subsystem is unavailable ▪ DS_EFAULT – Bad buffer address ▪ DS_EMSGSIZE – Message is too large to be sent all at once over the requested transport ▪ DS_EWOULDBLOCK – No free buffers are available for writing In addition to the above error codes, the following error codes can be returned in case the DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- Total number of bytes written, which could be less than the number of bytes specified (including partial write of a buffer).
- On error, it returns DSS_ERROR and places the value in *errno.
- An additional return value exists in the *errno variable. If this value is nonzero, a valid return value exists in this variable and should be handled accordingly.

Description

This function sends an iovcount number of bytes across all buffers over the TCP/UDP transport.

Provides the gather write variant of the dss_write() call that allows the application to write from noncontiguous buffers.

If the specified number of bytes to write is not equal to the number of bytes actually written, the call will return the number of bytes written. The application then needs to call dss_async_select() specifying DS_WRITE_EVENT. When the DS_WRITE_EVENT is asserted, the Application Callback function is invoked. The application should then use dss_getnextevent() to determine if DS_WRITE_EVENT is true. If it is true, the application can make another call to dss_writev() to send the remaining bytes.

Example

```
bytes_per_write = dss_writev(sock_fd,
                             iov,
                             iov_count,
                             &error_num);

if ( (bytes_per_write == DSS_ERROR) && (error_num == DS_EWOULDBLOCK) )
{
  rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
  dss_async_select(sock_fd,DS_WRITE_EVENT,
                   &error_num);
}
```

3.6.3 dss_sendto()

This function sends nbytes bytes in the buffer over the UDP transport. If the number of bytes to be sent is larger than that of the maximum UDP message size, none of the requested bytes in the buffer will be sent.

Parameters

```
sint15 dss_sendto (
                      sint15    sockfd,
                      const void    *buffer,
                      uint16    nbytes,
                      uint16    flags,
                      struct    sockaddr  *toaddr,
                      uint16    addrlen,
                      sint15    *errno
                      )
```

→	dss_sendto		
	→	sockfd	Socket descriptor
	→	buffer	Data buffer from which data is copied Buffer is owned by the application; socket layer does not change the buffer in any way; application is responsible for freeing the buffer
	→	nbytes	Number of bytes to be written to the socket
	→	flags	Supported values are: ▪ 0 – Special processing is not required for this packet ▪ MSG_EXPEDITE – Indicates that the packet should be sent on ACH or REACH if traffic channel is not UP ▪ MSG_FASTEXPEDITE – Indicates that the packet should be sent on REACH if traffic channel is not UP

	→	toaddr	Pointer to destination IP address and port number
			Related address data structures are defined as follows:
			
			`struct sockaddr`
			`{`
			`  uint8  sa_family;`
			`  char  sa_data[14];`
			`}`
			`struct in_addr`
			`{`
			`  uint32  s_addr;`
			`}`
			`  union`
			`  {`
			`  uint8  u6_addr8[16];`
			`  uint16  u6_addr16[8];`
			`  uint32  u6_addr32[4];`
			`  uint64  u6_addr64[2];`
			`  }in6_u;`
			`};`
			`struct sockaddr_in`
			`{`
			`  uint8  sin_family;`
			`  uint16  sin_port;`
			`  struct in_addr  sin_addr;`
			`  char  sin_zero[8];`
			`};`
			`struct sockaddr_in6`
			`{`
			`  uint8  sin_family;`
			`  uint16  sin_port;`
			`  uint32  sin6_flowinfo;`
			`  struct in6_addr  sin6_addr;`
			`  uint32  sin6_scope_id;`
			`};`

	→	addrlen	Length of toaddr

	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_EAFNOSUPPORT – Address family is not supported ▪ DS_EWOULDBLOCK – No free buffers are available for writing ▪ DS_ENETDOWN – Network subsystem is unavailable ▪ DS_EFAULT – Bad buffer address ▪ DS_EOPNOSUPPORT – Specified option is not supported; currently, this implementation does not support flags ▪ DS_ENOROUTE – No route to the destination found ▪ DS_EADDRREQ – Destination address is requested In addition to the above error codes, the following error codes can be returned in case the DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- Total number of bytes written, which could be less than the number of bytes specified
- On error, it returns DSS_ERROR and places the value in *errno.
- An additional return value exists in the *errno variable. If this value is nonzero, a valid return value exists in this variable and should be handled accordingly.

Description

This function sends nbytes bytes in the buffer over the UDP transport.

The first call to dss_sendto() does a routing lookup to determine the appropriate transmit interface. The result and the destination address of the first packet are then cached. Subsequent calls to dss_sendto() use the cached interface until the destination changes, thus triggering a new routing lookup.

If the specified number of bytes to write is not equal to the number of bytes actually written, the call returns DS_EWOULDBLOCK in the *errno variable. The application then needs to call dss_async_select() specifying DS_WRITE_EVENT. When the DS_WRITE_EVENT is asserted, the Application Callback function is invoked. The application should then use dss_getnextevent() to determine if DS_WRITE_EVENT is true. If it is true, the application can make another call to dss_sendto() to send the bytes.

Special considerations

This implementation does not support TCP sockets; therefore, any calls to dss_sendto() with a TCP socket will return an error.

Example

Consider sending datagrams to www.qualcomm.com, which is mapped to the class C IP address 192.35.156.11 and uses port 32768. The hexadecimal representation of this IP address is:

```
server_ip = 0xC0239C0B
```

And

```
port_num = 32768
```

The function call can then be used as in this example:

```
/*-------------------------------------------------
    Socket successfully opened. Fill in serv_addr struct.
    Remember to use network byte-ordering.
  -------------------------------------------------*/
  memset((char *) &serv_addr, 0, sizeof(serv_addr));
  serv_addr.sin_family      = AF_INET;
  serv_addr.sin_addr.s_addr = dss_htonl(server_ip);
  serv_addr.sin_port        = dss_htons(port_num);
  flags_are_unsupported     = 0;

  /*-------------------------------------------------
    Now send datagrams using dss_sendto().
  -------------------------------------------------*/
  MSG_MED("Sending datagrams to destination server",0,0,0);

  bytes_per_write = dss_sendto( sock_fd,
                                (byte *) buffer,
                                BYTES_TO_WRITE,
                                flags_are_unsupported,
                                 (struct sockaddr *)&serv_addr,
                                sizeof(serv_addr),
                                &error_num
                               );

  if ((bytes_per_write == DSS_ERROR)&&(error_num == DS_EWOULDBLOCK))
  {
    rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
    dss_async_select(sock_fd,DS_WRITE_EVENT,
                     &error_num);
  }
```

3.6.4 dss_sendmsg()

This function sends data from the msghdr struct over the socket-specified transport mechanism.

Parameters

```
sint15 dss_sendmsg (
                      sint15    sockfd,
                      struct    dss_msghdr     *msg,
                      uint16    flags,
                      sint15    *errno
                      )
```

→	dss_sendmsg		
	→	Sockfd	Socket descriptor

	→	dss_msghdr	It is the responsibility of the application to allocate the space for dss_msghdr.
			
			`struct dss_msghdr`
			`{`
			`  void  * msg_name;`
			`  uint16  msg_namelen;`
			`  struct iovec  * msg_iov;`
			`  uint16  msg_iovlen;`
			`  void  * msg_control;`
			`  uint16  msg_controllen;`
			`  int  msg_flags;`
			`};`
			
			msg_name is of type struct sockaddr.
			msg_namelen should be the size of the sockaddr being sent.
			msg_iov is of type:
			
			`struct iovec`
			`{`
			`  byte *iov_base;`
			`  uint16 iov_len;`
			`};`
			
			This buffer (iov_base) is owned by the application. It is the responsibility of the application to have enough buffer space allocated for at least iov_len bytes. See dss_writev for further description of the iovec struct and its usage.
			msg_iovlen represents the number of elements of msg_iov.
			msg_control is currently not supported for dss_sendmsg(). This should be NULL.
			msg_controllen is ignored in dss_sendmsg(). This should be 0.
			flags is currently not supported and should be set to 0.

	→	flags	Not supported – Must be set to zero

	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor was specified ▪ DS_EAFNOSUPPORT – Address family is not supported ▪ DS_EWOULDBLOCK – No data is available for reading ▪ DS_EADDRREQ – Destination address required ▪ DS_ENETDOWN – Network subsystem is unavailable ▪ DS_EFAULT – Bad buffer address ▪ DS_EMSGSIZE –Msg is too large to be sent all at once ▪ DS_EISCONN – If the socket is connected and the destination address is other than that to which it is connected ▪ DS_EOPNOSUPPORT – Specified option is not supported; this implementation does not support flags In addition to the above error codes, the following error codes can be returned in case the DSS_SO_ERROR_ENABLE socket option is enabled and there is a pending ICMP error for the socket: ▪ DS_ENETUNREACH – Network is unreachable ▪ DS_EHOSTUNREACH – Host is unreachable ▪ DS_EHOSTDOWN – Host is down ▪ DS_ENONET – Host is not on the network ▪ DS_EPROTO – Protocol error ▪ DS_EACCES – Access denied ▪ DS_ENOPROTOOPT – Protocol unreachable ▪ DS_ECONNREFUSED – Port unreachable ▪ DS_EMSGSIZE – Message too large ▪ DS_EOPNOTSUPP – Operation not supported

Return value

This function returns:

- Actual number of bytes written, which could be less than the number of bytes specified.
- On error, it returns DSS_ERROR and places the value in *errno.
- An additional return value exists in the *errno variable. If this value is nonzero, a valid return value exists in this variable and should be handled accordingly.

Description

This function is a common write function for all the socket output functions. The message header contains an array of scattered buffers, a socket descriptor, and destination address for unconnected UDP sockets. The function writes data from the scattered buffers over the transport specified by the socket descriptor.

If the specified number of bytes to write is not equal to the number of bytes actually written, the call will return the number of bytes written. The application then needs to call dss_async_select() specifying DS_WRITE_EVENT. When the DS_WRITE_EVENT is asserted, the Application Callback function is invoked. The application should then use dss_getnextevent() to determine if DS_WRITE_EVENT is true. If it is true, the application can make another call to dss_write() to send the remaining bytes.

Example

```
/*-------------------------------------------------------
  Now send datagrams using dss_sendmsg().
-------------------------------------------------------*/

bytes_per_write = dss_sendmsg( sock_fd,
                               &msghdr,
                               flags_are_unsupported,
                               &error_num
                             );

if ((bytes_per_write == DSS_ERROR)&&(error_num == DS_EWOULDBLOCK))
{
  rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
  dss_async_select(sock_fd,DS_WRITE_EVENT,
                   &error_num);
}
```

3.7 Termination

3.7.1 dss_shutdown()

This function causes all or parts of a full-duplex connection on a socket to be shut down.

Parameters

```
sint15 dss_shutdown (
                        sint15    sockfd,
                        uint16    how,
                        sint15    *errno
                        )
```

→	dss_shutdown		
	→	sockfd	Socket descriptor
	→	how	Type of shutdown: ▪ DSS_SHUT_WR – Disallow further receives ▪ DSS_SHUT_WR – Disallow further sends ▪ DSS_SHUT_RDWR – Disallow further receives and sends
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor is specified ▪ DS_EINVAL – Invalid operation, i.e., how parameter has invalid value ▪ DS_ENOMEM – Insufficient memory available to complete the operation ▪ DS_ENOTCONN – Socket is not connected

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in *errno

Description

This function causes all or parts of a full-duplex connection to be shut down. This call is supported only for connected sockets. Hence, the caller will get an error if dss_shutdown() is called for an unconnected TCP or UDP socket.

When dss_shutdown() is called for shutting down the read half, the received data queued on the socket is freed up and all subsequent incoming data is dropped. Furthermore, any subsequent calls to receive incoming data indicate a DS_EEOF error, but calls to send out data proceed normally.

When dss_shutdown() is called for shutting down the write half for a TCP socket, it causes TCP to initiate a graceful close, i.e., to send a FIN after sending all the queued-up outgoing data. When a similar call is made for a connected UDP socket, it does not result in any communication with the peer, and the socket is merely marked as unwilling to send out more data. After calling dss_shutdown() for a socket, any subsequent calls to send out data indicate a DS_ESHUTDOWN error, but calls to receive incoming data proceed normally.

When dss_shutdown() is called to shut down the read and write halves of the connection, the actions associated with shutting down the read half and the write half occur.

It is important to note that dss_shutdown() does not replace the functionality of the dss_close() call. dss_shutdown() deals with merely shutting down the data flow in one or both the directions and dss_close() should still be called to release the socket resources.

Example

```
ret_val = dss_shutdown(sock_fd, DSS_SHUT_RD, &error_num);

if (ret_val == DSS_SUCCESS)
{
  MSG_MED("Socket %d shut down for read-half”, sock_fd, 0, 0);
}
else
{
/* Bad error
 */
   MSG_ERROR("Could not shut down socket, error=%d”, error_num, 0,0);
}
```

3.7.2 dss_close()

This function closes a socket, performs all necessary cleanup of data structures, and frees the socket for reuse.

- For TCP sockets, if the socket is in the OPEN/OPENING state, this function initiates the active close for connection termination (graceful close). Otherwise it frees the socket resources immediately. Note that this is a nonblocking close.
- UDP sockets also need to call dss_close() to free the socket resource, making it available for reuse.

Parameters

```
sint15 dss_close (
                        sint15    sockfd,
                        sint15    *errno
                        )
```

→	dss_close		
	→	sockfd	Socket descriptor
	←	errno	Pointer to the returned error value that is passed by reference Mandatory to pass a non-NULL value to this parameter for successful operation The following are valid values for errno: ▪ DS_EBADF – Invalid socket descriptor is specified ▪ DS_EWOULDBLOCK – Operation would block

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in *errno

Description

This function performs a nonblocking close for a socket. It performs all necessary cleanup of data structures and frees the socket for reuse. For TCP sockets, if the socket is in a connected state, the initial call to dss_close() initiates the active close for connection termination and returns DS_EWOULDBLOCK. The client should then call dss_async_select(DS_CLOSE_EVENT). When it has finished its termination procedures (making the socket closable), DS_CLOSE_EVENT will be asserted. The application callback function is invoked to provide event notification, and the application can call dss_getnextevent() to determine which event occurred. Since the DS_CLOSE_EVENT will have been asserted, the application can call dss_close() again to free the socket for reuse. If the socket is in an unconnected state, the initial close call will free the resources for reuse and returns DSS_SUCCESS immediately.

UDP sockets also need to call dss_close() to clean up the socket and free it for reuse; however, the call would immediately return with DSS_SUCCESS.

Example

```
ret_val = dss_close(sock_fd,&error_num);

   if (ret_val == DSS_SUCCESS)
   {
      MSG_MED("Socket closed with no wait",0,0,0);
      ret_val = APP_SUCCESS;
   }
   else
   {
      if (error_num == DS_EWOULDBLOCK)
      {
         /* TCP close in progress
          */
         rex_clr_sigs(rex_self(),APP_SOCKET_CB_SIG);
         dss_async_select(sock_fd,DS_CLOSE_EVENT, &error_num);

         /* Application blocking control layer
          */
         application_wait(APP_SOCKET_CB_SIG);

         /* Have to close again. Should succeed.
          */
         ret_val = dss_close(sock_fd,&error_num);
         ASSERT(ret_val == DSS_SUCCESS);
      }
      else
      {
         /* Bad error....should always be able to close
          */
         ERR("Could not close socket",0,0,0);
         ret_val = DSS_ERROR;
      }
   }
```

3.7.3 dsnet_release_handle()

NOTE: dsnet_release_handle() replaces the previous function name, dss_close_netlib().

This function closes the network library for the application.

Parameters

```
sint15 dsnet_release_handle (
                                  sint15    nethandle,
                                  sint15    *errno
                                  )
```

→	dsnet_release_handle		
	→	nethandle	Network handle
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADAPP – Invalid network handle specified ▪ DS_SOCKEXIST – Allocated sockets exist ▪ DS_ENETEXIST – Network subsystem is still established

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in *errno

Description

This function closes the network library for the application. All sockets must have been closed for the application prior to closing. If this is the last remaining application, the network subsystem (PPP/traffic channel) must have been brought down prior to closing the network library. dsnet_release_handle() is essentially a blocking call, i.e., there are no events associated with or signals set when the library is closed.

This function is called from the context of the socket's client task.

Example

```
/*-----------------------------------------------------------
   Now close the network library.
-----------------------------------------------------------*/
MSG_MED("Closing sockets library",0,0,0);
ret_val = dsnet_release_handle(socket_nethandle,
                           &error_num
                           );
if (ret_val == DSS_ERROR)
{
   /* We have an error. We can look at error_num to see what
    * it was and put value to DIAG.
    */

  ERR("Could not close network library: %d",error_num,0,0);
  return;
}
```

3.8 Administration

3.8.1 Socket options

Table 3-2 lists the socket options currently supported. The flag column indicates whether the socket option can be used as a Boolean flag, with 0 meaning FALSE, and any other integer meaning TRUE.

Table 3-2 Currently supported socket options

Socket option name	Description	Level	Default value	Data type	Option value size	Min	Max	Flag
DSS_IP_TTL	Time-to-live	DSS_IPPROTO_IP	255	int	sizeof (int)	1	255	
DSS_SO_RCVBUF	Change the receive buffer size for the socket	DSS_SO_SOCK	16384 (non HDR) 46080 (HDR)	int	sizeof (int)	536	196608	
DSS_SO_SNDBUF	Change the send buffer size for the socket	DSS_SO_SOCK	10240	int	sizeof (int)	536	32768 36864 (REL A)	
DSS_SO_LINGER	Linger on close if there is data to send	DSS_SO_SOCK	(0,0)	dss_so_linger_type	sizeof (dss_so_linger_type)	Same as default value	(1,NA)	
DSS_TCP_MAXSEG	Change the TCP maximum segment size	DSS_IPPROTO_TCP	1460	int	sizeof (int)	Same as default value	18432	
DSS_SO_SDB_ACK_CB	Register callback that is invoked on receiving an SDB/DOS data ACK	DSS_SOCK	0	dss_so_sdb_ack_cb_type	sizeof(dss_so_sdb_ack_cb_type)	0	0	
DSS_TCP_NODELAY	Disable Nagle's algorithm	DSS_IPPROTO_TCP	0	int	sizeof (int)	0	1	*
DSS_SO_KEEPALIVE	Enables periodic probes on idle TCP connection	DSS_SO_SOCK	0	int	sizeof(int)	0	1	*
DSS_TCP_DELAYED_ACK	Reduce the number of ACKs sent to half	DSS_IPPROTO_TCP	0	int	Sizeof(int)	0	1	*
DSS_TCP_SACK	Enables TCP selective ACK option	DSS_IPPROTO_TCP	0	int	sizeof(int)	0	1	*
DSS_TCP_TIME_STAMP	Enables TCP time stamp option	DSS_IPPROTO_TCP	0	int	Sizeof(int)	0	1	*

Socket option name	Description	Level	Default value	Data type	Option value size	Min	Max	Flag
DSS_BCMCS_JOIN	Join multicast group	DSS_IPPROTO_IP	0	dss_so_ip_addr_type	Sizeof(dss_so_ip_addr_type)	0	0	
DSS_BCMCS_LEAVE	Leave multicast group	DSS_IPPROTO_IP	0	dss_so_ip_addr_type	Sizeof(dss_so_ip_addr_type)	0	0	
DSS_IP_RECVIF	Retrieve Rx iface of packet	DSS_IPPROTO_IP	0	int	Sizeof(int)	0	1	*
DSS_SO_NETPOLICY	Set/retrieve network policy	DSS_SO_SOCK	0	dss_net_policy_info_type	sizeof(dss_net_policy_info_type)	0	1	
DSS_IP_TOS	Set TOS for outgoing IP packets	DSS_IPPROTO_IP	0	int	sizeof(int)	0	255	
DSS_SO_CB_FCN	Set the socket Callback function	DSS_SOCK	0	dss_sock_cb_fcn_type	sizeof(dss_sock_cb_fcn_type)	0	0	
DSS_SO_ERROR_ENABLE	Enable retrieving of ICMP errors using SO_ERROR	DSS_SOCK	0	int	sizeof(int)	0	1	*
DSS_SO_ERROR	Get ICMP error on the socket	DSS_SOCK	0	Int	sizeof(int)	0	255	
DSS_IP_RECVERR	Enable retrieving of ICMP error info with dss_recvmsg	DSS_IPPROTO_IP	0	int	sizeof(int)	0	1	*
DSS_IPV6_RECVERR	Enable retrieving of ICMPv6 error info with dss_recvmsg	DSS_IPPROTO_IP	0	int	sizeof(int)	0	1	*
DSS_SO_LINGER_RESET	Linger on close and reset on timeout	DSS_SOCK	(0,0)	dss_so_linger_type	sizeof (dss_so_linger_type)	Same as default value	(1,NA)	
DSS_IPV6_TCLASS	Set the traffic class for the IPv6 socket	DSS_IPPROTO_IPV6	0	int	sizeof(int)	0	255	
DSS_SO_REUSEADDR	Enable socket port reuse	DSS_SOCK	0	int	sizeof(int)	0	1	*
DSS_SO_DISABLE_FLOW_FWDING	Disables routing data on best effort flow when QoS is not available	DSS_SOCK	0	int	sizeof(int)	0	1	*
DSS_TCP_FIONREAD	Gets the TCP receive queue length	DSS_IPPROTO_TCP	N/A	uint32	sizeof(uint32)	0	0xFFFFFFFF	
DSS_IP_ADD_MEMBERSHIP	Request to join a IPv4 multicast group.	DSS_IPPROTO_IP	0	dss_ip_mreqn	sizeof(dss_ip_mreqn)	0	0	

Socket option name	Description	Level	Default value	Data type	Option value size	Min	Max	Flag
DSS_IP_DROP_MEMBERSHIP	Request to leave a IPv4 multicast group	DSS_IPPROTO_IP	0	dss_ip_mreqn	sizeof(dss_ip_mreqn)	0	0	
DSS_IPV6_ADD_MEMBERSHIP	Requests to join a IPv6 multicast group	DSS_IPPROTO_IPV6	0	dss_ipv6_mreqn	sizeof(dss_ipv6_mreqn)	0	0	
DSS_IPV6_DROP_MEMBERSHIP	Request to leave a IPv6 multicast group	DSS_IPPROTO_IPV6	0	dss_ipv6_mreqn	sizeof(dss_ipv6_mreqn)	0	0	
DSS_IP_MULTICAST_TTL	Specifies the time-to-live value for multicast datagrams sent through this socket	DSS_IPPROTO_IP	0	Int	Sizeof(int)	1	255	
DSS_SO_FIONREAD	Gets TCP/UDP/ICMP socket recv Q length	DSS_SOCK	0	Int	Sizeof(int)	0	262144	
DSS_SO_SKIP_ROUTE_SCOPE_CHECK	Enable/Disable route scope check skip option	DSS_SOCK	0	Int	Sizeof(int)	0	0x7FFFFFFF	
DSS_TCP_MAX_BACKOFF_TIME	Sets TCP backoff timer	DSS_IPPROTO_TCP	120000	Int	Sizeof(int)	0	120000	
DSS_IP_RECVTTL	Enable retrieving of TTL or Hop Limit info of the received packet	DSS_IPPROTO_IP	0	int	Sizeof(int)	0	1	*
DSS_IP_RECVECN	Enable retrieving of ECN info of the received packet	DSS_IPPROTO_IP	0	int	Sizeof(int)	0	1	*
DSS_UDP_ENCAPS	Enable UDP encapsulation	DSS_IPPROTO_UDP	0	int	Sizeof(int)	0	1	*

Following is a brief description of the socket options that are nonstandard or that differ from the standard BSD behavior.

DSS_SO_SILENT_CLOSE

The usage of this socket option is deprecated. All the applications should use the DSS_SO_LINGER option with a zero timeout value instead of the DSS_SO_SILENT_CLOSE socket option.

DSS_SO_SDB_ACK_CB

This socket option is applicable only on CDMA networks. This socket option is used by applications using the Short Data Burst (SDB) or Data over Signaling (DoS) features. It is used to retrieve or register a callback that is invoked when the mobile receives an SDB/DoS data ACK from the base station. The base station sends an SDB/DoS ACK for each SDB or DoS packet that the base station receives from the mobile. The callback dss_so_sdb_ack_cb_type is defined as follows:

```
typedef void (*dss_so_sdb_ack_cb_fcn) (
                  sint15                          sockfd,
                  dss_sdb_ack_status_info_type    * sdb_ack_info,
                  void                            * userdata
                  )
```

→	dss_so_sdb_ack_cb_fcn		
	→	sockfd	Socket using which SDB/DoS packet is sent by the application

	→	sdb_ack_info	Additional information about the SDB/DoS ack.
			dss_sdb_ack_status_info_type is defined as:
			
			`typedef struct dss_sdb_ack_status_info_type`
			`{`
			`  uint32 overflow;`
			`  dss_sdb_ack_status_enum_type status;`
			`} dss_sdb_ack_status_info_type;`
			
			Overflow is set to a nonzero value, if the number of outstanding SDB/DoS packets, i.e., the packets for which mobile is still waiting for an ACK, exceeds the number the mobile can handle.
			Status is of type dss_sdb_ack_status_info_type and the possible values are:
			▪ DSS_SDB_ACK_NONE – Status is not available
			▪ DSS_SDB_ACK_OK – Packet is sent successfully
			▪ DSS_SDB_ACK_HOLD_ORIG_RETRY_TIMEOUT – Hold orig timer expired and hence failed to send the packet
			▪ DSS_SDB_ACK_HOLD_ORIG – Unable to process the packet because hold orig is true
			▪ DSS_SDB_ACK_NO_SRV – Failed to send the packet due to lack of service
			▪ DSS_SDB_ACK_ABORT – Aborted the operation
			▪ DSS_SDB_ACK_NOT_ALLOWED_IN_AMPS – SDB/DS is not supported in analog mode
			▪ DSS_SDB_ACK_NOT_ALLOWED_IN_HDR – SDB/DS is not supported when in a HDR call
			▪ DSS_SDB_ACK_L2_ACK_FAILURE – Failed to receive Layer 2 ACK
			▪ DSS_SDB_ACK_OUT_OF_RESOURCES – Unable to process the packet because of lack of resources
			▪ DSS_SDB_ACK_ACCESS_TOO_LARGE – Packet is too big to be sent over access channel
			▪ DSS_SDB_ACK_DTC_TOO_LARGE – Packet is too big to be sent over DTC
			▪ DSS_SDB_ACK_ACCT_BLOCK – Access channel is blocked for traffic based on service option
			▪ DSS_SDB_ACK_L3_ACK_FAILURE – Failed to receive Layer 3 ACK
			▪ DSS_SDB_ACK_OTHER – Failed for some other reason

	→	userdata	Application specified user data

If an application writes more than one SDB/DoS packet simultaneously, it is not possible to correlate the ACK with one of the packets. This is because there is no provision to uniquely identify each and every packet. So if the application wants to check if the ACK is received for a particular SDB/DOS packet, it is recommended that the application have at most only one outstanding SDB/DOS packet at any time. For more details on DoS, see [**Error! Reference source not found.**].

DSS_TCP_MAXSEG

This socket option allows us to fetch or set the Maximum Segment Size (MSS) for a TCP connection. The value returned is the maximum amount of data that the mobile's TCP will send to the other end in a single segment. If this value is fetched before the socket is connected, the value returned will be the default value and is the value that the mobile announces in its SYN segment. The actual MSS value of a connection is the minimum user-specified MSS value, the

Maximum Transmission Unit (MTU) of the link, and the peer's advertised MSS. Applications must set this value to less than 1, which is the current MSS value.

DSS_TCP_SACK

If set, this option enables the TCP Selective Acknowledgement (SACK) option (see [**Error! Reference source not found.**]). The purpose of SACK is to inform the sender about all segments that have arrived successfully. We generate SACK if this option is set by the application and we receive a SACK_PERMITTED option from the peer in its SYN segment. Currently, we support generating only selective acknowledgements.

DSS_TCP_TIMESTAMP

If set, this option enables the TCP timestamp option (see [**Error! Reference source not found.**]). Enabling the timestamp option for a connection yields accurate measurements of RTT and result in an accurate measurement of Smooted Round-Trip Time (SRTT), based on every segment sent instead of one segment per window. This option is recommended for connections with large receive windows and changing traffic conditions as RTT estimates based on one segment per window will not yield an accurate RTT estimate.

DSS_BCMCS_JOIN and DSS_BCMCS_LEAVE

These socket options have been deprecated. Applications should no longer use these and must begin using the new API. See Sections 0 to 0 on how to use the new Multicast IOCTLs: DSS_IFACE_IOCTL_MCAST_JOIN, DSS_IFACE_IOCTL_MCAST_LEAVE, DSS_IFACE_IOCTL_MCAST_JOIN_EX, DSS_IFACE_IOCTL_MCAST_LEAVE_EX and DSS_IFACE_IOCTL_MCAST_REGISTER_EX.

DSS_SO_NETPOLICY

This option allows an application to retrieve or set the network policy using the dss_net_policy_info_type. The policy can be set only when using dssocket2(). If the application has created a socket using dssocket(), dss_setsockopt will return DS_EOPNOTSUPP. Instead, the application should use dsnet_set_policy().

DSS_SO_CB_FCN

This socket option is used by the applications to set the socket callback function. It is supported only for sockets created using dss_socket2(). If the application has created a socket using dss_socket(), dss_setsockopt will return DS_EOPNOTSUPP.

DSS_SO_ERROR_ENABLE

This socket option is used by the applications to enable retrieving ICMP errors received on the socket using the DSS_SO_ERROR socket option.

DSS_SO_ERROR

This socket option is used by the applications to retrieve ICMP errors received on the socket. The integer returned by dss_getsockopt() is the latest pending ICMP error received on this socket in the form of a relevant error code. Only dss_getsockopt() is supported for this socket option. A dss_setsockopt() on this socket option returns DS_ENOPROTOOPT. This socket option is usable only when the DSS_SO_ERROR_ENABLE socket option is enabled. Otherwise, it returns DS_EFAULT.

DSS_IP_RECVERR

This socket option is used by the applications to retrieve extended information on ICMP errors received on the socket. This socket option is supported only for datagram sockets. dss_setsockopt() returns DS_EINVAL if the application tries to enable this socket option on a stream socket. When enabled on a datagram socket, all received ICMP errors are enqueued on a per-socket error queue. The maximum number of errors enqueued per socket is five. If on a socket, and five errors have already been enqueued, when a new error is received, the oldest error in the queue is dropped and the new one is enqueued. When the user receives an ICMP error due to a socket operation, the error can be retrieved by calling dss_recvmsg() with the MSG_ERRQUEUE flag set. The extended error information is returned in the ancillary data portion of struct dss_msghdr in the form of struct dss_cmsghdr with cmsg_level set to DSS_IPPROTO_IP and cmsg_type set to DSS_IP_RECVERR. The data portion of struct dss_cmsghdr contains the ICMP error information in the form of dss_sock_extended_err structure (shown below) followed by the sockaddr of the node that sent this ICMP error. The port number of the sockaddr is set to 0. The payload of the original packet that caused the error is passed as normal data via msg_iov. The original destination of the datagram that caused the error is returned via msg_name.

The control message for extended error information contains the dss_sock_extended_err structure:

```
typedef struct
{
  uint32 ee_errno;         /* error number               */
  uint8  ee_origin;        /* where the error originated */
  uint8  ee_type;          /* ICMP type                  */
  uint8  ee_code;          /* ICMP code                  */
  uint8  ee_pad;           /* padding                    */
  uint32 ee_info;          /* additional information     */
  uint32 ee_data;          /* other data                 */
  /* More data may follow                               */
} dss_sock_extended_err;
```

ee_origin is set to DSS_SO_EE_ORIGIN_ICMP for received ICMP errors. ee_info contains discovered MTU for DS_EMSGSIZE errors.

DSS_IPV6_RECVERR

This socket option is used by the applications to retrieve extended information on ICMPv6 errors received on the socket. This socket option is supported only for datagram sockets. The function dss_setsockopt() returns DS_EINVAL if the application tries to enable this socket option on a stream socket. When enabled on a datagram socket, all received ICMPv6 errors are enqueued on a per-socket error queue. The maximum number of errors enqueued per socket is five. If five errors have already been enqueued on a socket and a new error is received, the oldest error in the queue is dropped and the new one is enqueued. When the user receives an ICMPv6 error due to a socket operation, the error can be retrieved by calling dss_recvmsg() with the MSG_ERRQUEUE flag set. The extended error information is returned in the ancillary data portion of struct dss_msghdr in the form of struct dss_cmsghdr with the cmsg_level set to DSS_IPPROTO_IPV6 and cmsg_type set to DSS_IPV6_RECVERR. The data portion of struct dss_cmsghdr contains the ICMPv6 error information in the form of dss_sock_extended_err structure (shown below) followed by the sockaddr of the node that sent this ICMPv6 error. The port number of the sockaddr is set to 0. The payload of the original packet that caused the error is passed as normal data via msg_iov. The original destination of the datagram that caused the error is returned via msg_name.

The control message for extended error information contains the following dss_sock_extended_err structure:

```
typedef struct
{
  uint32 ee_errno;         /* error number               */
  uint8  ee_origin;        /* where the error originated */
  uint8  ee_type;          /* ICMPv6 type                */
  uint8  ee_code;          /* ICMPv6 code                */
  uint8  ee_pad;           /* padding                    */
  uint32 ee_info;          /* additional information     */
  uint32 ee_data;          /* other data                 */
  /* More data may follow                               */
} dss_sock_extended_err;
```

ee_origin is set to DSS_SO_EE_ORIGIN_ICMPv6 for received ICMPv6 errors. ee_info contains the discovered MTU for DS_EMSGSIZE errors.

DSS_SO_LINGER_RESET

This socket option is used to set the “linger reset” timeout value for a TCP socket. If the timeout value is set to zero the TCP connection is aborted and a reset message is sent when dss_close is called. If the timeout value is set to a value greater than zero then the socket lingers trying to do a graceful close until the timer expires. Application will get a close event when we are able to do a graceful socket close or when the “linger reset” timer expires. Upon a timeout the connection is aborted and a reset message is sent to the peer.

DSS_SO_RCVBUF

This socket option is used by the application to configure the sockets receive buffer. For TCP sockets this socket option also specifies the receive window of the TCP socket. If the receive window is configured to a value greater than 64k then TCP window scaling option is enabled.

DSS_IPV6_TLCASS

This socket option is used by the application to configure the socket’s IPv6 traffic class. The traffic class specifies the classes and priorities of the IPv6 packets. This replaces the TOS field from IPv4 headers.

DSS_SO_REUSEADDR

This socket option is used to allow sockets to reuse a port that is already bound by another socket. With this socket option, multiple sockets can bind to a port number using the same multicast address or INADDR_ANY. Only one of the sockets binding to this port number can use INADDR_ANY, while other sockets must bind to a valid multicast address. Currently, this socket option is supported only for multicast UDP sockets. The primary use of this socket option is to enable multiple sockets to simultaneously listen to a multicast stream.

DSS_SO_DISABLE_FLOW_FWDING

If a packet needs QoS and if a QoS instance is not activated, by default, AMSS forwards the packet on to a best-effort service flow. See Section 6.1.2 for more details. Although this is desirable for most applications, some applications, such as VoIP, want to send data only when QoS is available. Hence, this socket option is introduced to disable forwarding data on to best effort service flow when QoS instance is not activated.

Applications are required to keep track of QoS status and activate QoS instance when QoS is not available. Otherwise, the application may run into the following infinite loop: application writes data, AMSS returns DS_EWOULDBLOCK because QoS is not available and the socket option is set, application asserts for WRITE_EV, AMSS posts WRITE_EV in order to wake up the application.

DSS_TCP_FIONREAD

This socket option is used by the application to check for the amount of data (in bytes) currently queued in the TCP receive queue for the socket. This allows the application to read all the available data at once from the socket. It is however possible that after an application gets the TCP receive queue size, and before the read can be performed, more data is enqueued in the TCP receive queue, thus requiring additional reads.

DSS_IP_ADD_MEMBERSHIP

This socket option is used to join an IPv4 multicast group using the Internet Group Management Protocol (IGMP) on a specified local interface. Only one multicast address can be joined per socket.

DSS_IP_DROP_MEMBERSHIP

This socket option is used to leave an IPv4 multicast group that was previously joined by this socket using the socket option DSS_IP_ADD_MEMBERSHIP.

DSS_IPV6_ADD_MEMBERSHIP

This socket option is used to join an IPv6 multicast group using Multicast Listener Discovery (MLD) on a specified local interface. Only one multicast address can be joined per socket.

DSS_IPV6_DROP_MEMBERSHIP

This socket option is used to leave an IPv6 multicast group that was previously joined using the socket option DSS_IPV6_ADD_MEMBERSHIP.

DSS_IP_MULTICAST_TTL

This socket option is used to set the IP TTL value for the multicast packets going out on the specified socket.

DSS_SO_FIONREAD

This socket option is used to get TCP/UDP/ICMP socket receive queue length.

DSS_SO_SKIP_ROUTE_SCOPE_CHECK

This socket option is used to skip the route scope check for incoming data. Clients can use this option to capture packets coming on any interface and destined to the same port. Value of 0 indicates route scope check will be perfomed and value of 0x7FFFFFF will skip the route scope check.

DSS_TCP_MAX_BACKOFF_TIME

This socket option is used to set the maximum TCP back-off timer. Current default timer value is 2 seconds; to set the timer back to default value from a specified value, clients must pass 0 as the option.

DSS_IP_RECVTTL

This socket option is used by the application to retrieve TTL or Hop limit information from the received packet. This socket option is supported only for datagram sockets. TTL or Hop limit

value can be retrieved by calling dss_recvmsg(). TTL or Hop Limit information is returned in the ancillary data portion of struct dss_msghdr in the form of struct dss_cmsghdr with cmsg_type set to DSS_IP_RECVTTL. The data portion of struct dss_cmsghdr contains the TTL or Hop limit value.

DSS_IP_RECVECN

This socket option is used by the application to retrieve ECN information from the received packet. This socket option is supported only for datagram sockets. ECN value can be retrieved by calling dss_recvmsg(). ECN information is returned in the ancillary data portion of struct dss_msghdr in the form of struct dss_cmsghdr with cmsg_type set to DSS_IP_RECVECN. The data portion of struct dss_cmsghdr contains the ECN value.

DSS_UDP_ENCAPS

This socket option can be used by the application to use the socket for UDP encapsulation. This socket option must be set before the socket is bound to ephemeral port (i.e. dss_bind with port number 0).

3.8.2 dss_setsockopt()

This function sets the socket options associated with a socket.

Parameters

```
sint15 dss_setsockopt(
                      int sockfd,
                      int level,
                      int optname,
                      void    *optval,
                      uint32  *optlen,
                      sint15  *dss_errno
                      )
```

→	dss_setsockopt		
	→	sockfd	Socket descriptor
	→	level	Number of the protocol that controls the option. See Table 3-2 for the valid level of a specified socket option
	→	optname	Socket option name. See Table 3-2 for all supported socket options
	→	optval	Value of the socket option to be set
	→	optlen	Size of the socket option value passed in
	←	dss_errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor is specified ▪ DS_ENOPROTOOPT – Option requested is unknown at the level indicated ▪ DS_EINVAL – Invalid option name or option value ▪ DS_ENETINPROGRESS – Network subsystem establishment is currently in progress ▪ DS_EFAULT – Invalid buffer or argument

Return value

This function returns:

- DSS_ERROR and places the error condition value in *errno.
- On success, DSS_SUCCESS is returned.

Description

This function manipulates the socket options associated with a socket. See Table 3-2 for a list of currently supported socket options. In the case of listening TCP sockets, certain socket options will take effect only if they are set after the call to dss_listen() but before the TCP connection is established. For these socket options, dss_setsockopt() should be called on the listening socket descriptor before the TCP connection is established. These socket options are DSS_SO_RCVBUF, DSS_TCP_MAXSEG, DSS_TCP_SACK, and DSS_TCP_TIMESTAMP. For other socket options, the dss_setsockopt() should be called on the socket descriptor returned after the TCP connection has been accepted. In the future, these constraints will be removed.

Example

```
ret_val = dss_setsockopt(sock_fd,
                         DSS_IPPROTO_IP,
                         DSS_IP_TTL,
                         &val,
                         &len,
                         &error_num);
```

3.8.3 dss_getsockopt()

This function returns the socket options associated with a socket.

Parameters

```
sint15 dss_getsockopt(
                        int sockfd,
                        int level,
                        int optname,
                        void     *optval,
                        uint32   *optlen,
                        sint15   *dss_errno
                        )
```

→	dss_getsockopt		
	→	sockfd	Socket descriptor
	→	level	Number of the protocol that controls the option. See Table 3-2 for the valid level of a specified socket option
	→	optname	Socket option name. See Table 3-2 for all supported socket options
	←	optval	Identifies a buffer in which the value of the queried socket option value is to be returned

	↔	optlen	Value result parameter initially containing the size of the buffer pointed to by optval and modified on return to indicate the actual size of the value returned
	←	dss_errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor is specified ▪ DS_ENOPROTOOPT – Option requested is unknown at the level indicated ▪ DS_EINVAL – Invalid option name or option value ▪ DS_ENETINPROGRESS – Network subsystem establishment is currently in progress ▪ DS_EFAULT – Invalid buffer or argument

Return value

- On error, DSS_ERROR is returned and the error condition value is placed in *errno.
- On success, DSS_SUCCESS is returned; value and length of the queried socket option are also returned in *optval and *optlen respectively.

Description

This function returns the value and length of an option associated with an open socket. See Table 3-2 for a list of socket options currently supported.

Example

```
ret_val = dss_getsockopt(sock_fd,
                         DSS_IPPROTO_IP,
                         DSS_IP_TTL,
                         &val,
                         &len,
                         &error_num);
```

3.8.4 dss_getsockname()

This function returns the current local address, in network byte order, assigned to the specified socket.

Parameters

```
sint15 dss_getsockname (
                         sint15 sockfd,
                         struct sockaddr *addr,
                         uint16 *addrlen,
                         sint15 *errno
                         )
```

→	dss_getsockname		
	→	sockfd	Socket descriptor

	←	`addr`	Pointer to a memory block, allocated by the caller, containing the current local address of the socket. The specific address type depends on the underlying protocol.
	↔	`addrlen`	Pointer to a memory block, allocated by the caller, containing the length of the addr parameter. addrlen is initialized to indicate the amount of space provided by addr; on return, it contains the actual size of the address returned
	←	`errno`	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor is specified ▪ DS_EFAULT – Invalid memory address

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in *errno

Description

This function returns the current local address, in network byte order, assigned to the specified socket. The addrlen parameter should be initialized to indicate the amount of space provided by addr. On return, it contains the actual size of the address returned. The address is truncated if the supplied buffer is too small and the value returned in addrlen indicates the address length prior to truncation.

Example

```
struct sockaddr addr;
struct sockaddr_in* p_inaddr;
uint16 addrlen;
/*------------------------------------------------------------
   Obtain the local address of a IPv4 socket
------------------------------------------------------------*/
addrlen = sizeof(addr);
ret_val = dss_getsockname(sockfd,
                          &addr
                         &addrlen
                         &error_num
                        );
if (ret_val == DSS_ERROR)
{
   /* We have an error. We can look at error_num to see what
    * it was and send it to DIAG.
    */

  ERR("Could not get socket name: %d",error_num,0,0);
  return;
}
else
{
  p_inaddr = (struct sockaddr_in *) addr;
  MSG_HIGH("family=%d, port=%d, addr=0x%x",
           p_inaddr->sin_family, p_inaddr->sin_port, p_inaddr-
      >sin_addr.s_addr);
}
```

3.8.5 dss_getpeername()

This function returns the address of the peer, in network byte order, connected to the specified socket.

Parameters

```
sint15 dss_getpeername (
                        sint15 sockfd,
                        struct sockaddr *addr,
                        uint16 *addrlen,
                        sint15 *errno
                        )
```

→	dss_getsockname		
	→	sockfd	Socket descriptor
	←	addr	Pointer to a memory block, allocated by the caller, containing the address of the peer connected to the socket. The specific address type depends on the underlying protocol; currently only Pv4 is supported.
	↔	addrlen	Pointer to a memory block, allocated by the caller, containing the length of the addr parameter. addrlen is initialized to indicate the amount of space provided by addr; on return, it contains the actual size of the address returned
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADF – Invalid socket descriptor is specified ▪ DS_EFAULT – Invalid memory address ▪ DS_ENOTCONN – Socket is not connected

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in *errno

Description

This function returns the address of the peer, in network byte order, connected to the specified socket. The addrlen parameter should be initialized to indicate the amount of space provided by addr. On return, it contains the actual size of the address returned. The address is truncated if the supplied buffer is too small and the value returned in addrlen indicates the address length prior to truncation.

Example

```
  struct sockaddr addr;
  struct sockaddr_in* p_inaddr;
  uint16 addrlen;
  /*-------------------------------------------------------------
     Obtain the local address of an IPv4 socket
  -------------------------------------------------------------*/
  addrlen = sizeof(addr);
  ret_val = dss_getpeername(sockfd,
                            &addr
                            &addrlen
                            &error_num
                           );
  if (ret_val == DSS_ERROR)
  {
     /* We have an error. We can look at error_num to see what
      * it was and send it to DIAG.
      */

    ERR("Could not get the peer name: %d",error_num,0,0);
    return;
  }
  else
  {
    p_inaddr = (struct sockaddr_in *) addr;
    MSG_HIGH("family=%d, port=%d, addr=0x%x",
             p_inaddr->sin_family, p_inaddr->sin_port, p_inaddr-
>sin_addr.s_addr);
  }
```

3.9 DNS

3.9.1 dss_dns_create_session()

This function creates a session for performing DNS lookup.

Parameters

```
dss_dns_session_mgr_handle_type dss_dns_create_session(

                                dss_dns_cback_f_type    *app_cback_f_ptr,
                                void                    *user_data_ptr,
                                int16                   *dss_errno
)
```

→	dss_ dns_create_session		

	→	app_cback_ f_ptr	Application callback function; the callback function is stored in the session control block and is called whenever the dss_dns_get_addrinfo or dss_dns_get_nameinfo query is complete; the callback function is of type:
			`typedef void (*dss_dns_cback_f_type)`
			`(`
			`dss_dns_session_mgr_handle_type session_handle,`
			`dss_dns_query_handle_type query_handle,`
			`dss_dns_api_type_enum_type api_type,`
			`uint16 num_records,`
			`void * user_data_ptr,`
			`int16 dss_errno`
			`);`

	←	user_data_ ptr	User data passed in with callback
	←	dss_errno	Error code returned in case of error; valid values are: ▪ DS_EFAULT – NULL callback info provided. ▪ DS_ENOMEM – Maximum number of sessions reached.

Return value

This function returns:

- On success, the session handle of the session created
- On error, DSS_DNS_SESSION_INVALID_HANDLE

Description

The application calls this function to create a session for DNS lookup. The session contains a set of configuration information that directs the behavior of the DNS subsystem. The application must set the callback information while creating the session. After creating a session, an application can make calls to dss_dns_get_addrinfo and dss_dns_get_nameinfo functions for DNS lookup.

Example

```
int16                           dss_errno = 0;
int32                           cb_user_data = 0;
dss_dns_session_mgr_handle_type  session_handle;

/*-------------------------------------------------------------------
  We create a session with cback_fcn callback function.
-------------------------------------------------------------------*/
session_handle = dss_dns_create_session( cback_fcn,
                                         (void *) &cb_user_data,
                                         &dss_errno );

if( DSS_DNS_SESSION_INVALID_HANDLE == session_handle )
{
  MSG_ERROR("Session could not be created error %d", dss_errno, 0, 0);
}
```

3.9.2 dss_dns_set_config_params()

This function is used to set various configuration parameters for a given session.

Parameters

```
int16 dss_dns_set_config_params(
                dss_dns_session_mgr_handle_type   session_handle,
                dss_dns_config_params_enum_type   param_name,
                void                              *param_val_ptr,
                uint32                            param_len,
                int16                             *dss_errno
)
```

→	dss_dns_set_config_params		
	→	session_handle	Session control block identifier

	→	param_name	Type of the parameter to be set; this is an enum of type:
			`typedef enum dss_dns_config_params_enum_type`
			`{`
			`DSS_DNS_CONFIG_PARAMS_MIN,`
			`DSS_DNS_CONFIG_PARAMS_IFACE_ID,`
			`DSS_DNS_CONFIG_PARAMS_USE_CACHE_RESULTS,`
			`DSS_DNS_CONFIG_PARAMS_ADD_TO_CACHE,`
			`DSS_DNS_CONFIG_PARAMS_MAX`
			`} dss_dns_config_params_enum_type;`
			▪

	→	param_val_ptr	Pointer where the new value for the parameter is stored
	→	param_len	Length of the parameter; the caller specifies the length of the param_val_ptr argument that should be read
	←	dss_errno	Error code returned to the application in case of an error; valid values are: ▪ DS_EBADF – Invalid session handle ▪ DS_EFAULT – Invalid arguments

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error code value is set in dss_errno

Description

The application calls this function to set a given configuration parameter (specified by the param_name argument) to a given value (specified by param_val_ptr argument). The parameters in the following sections can be set using this function.

3.9.2.1 DSS_DNS_CONFIG_PARAMS_IFACE_ID

This option allows the application to set the interface identifier to be used for DNS lookup. This mode of operation is preferred when the application has already brought up an interface and wants to use that interface to do the DNS query. If this option has not been set, then a network policy with policy flag of DSS_IFACE_POLICY_UP_PREFERRED will be used to bring up the network and perform DNS lookup.

The data type of param_val_ptr argument should be dss_iface_id_type. param_len argument should be sizeof(dss_iface_id_type).

3.9.2.2 DSS_DNS_CONFIG_PARAMS_ROUTEABLE_IFACE_MODE

The embedded application can specify whether to perform the DNS lookup when the laptop call is active. To do so, the application needs to specify the routable interface mode configuration parameter. The param_val_ptr argument for this option is a Boolean pointer and the param len argument should be sizeof(boolean). This option can be set only if the iface ID configuration parameter for the session is set.

It is recommended that the application use DSS_DNS_CONFIG_PARAMS_NET_POLICY instead of using this option because, if the profile number of the routable interface does not match the default sockets profile number, the route lookup would fail, causing the DNS lookup to fail.

3.9.2.3 DSS_DNS_CONFIG_PARAMS_NET_POLICY

This option can be used by the application to set the network policy while doing the DNS lookup. If the network policy option is set in the DNS session, the DNS resolver would try to use the same network policy to send the DNS query. It would create a socket2 with the specified network policy and send out the query. Note that if this option is set, the DNS resolver will not try to bring up any interface.

This option can be used while a DNS lookup needs to be performed when a laptop call is active. The embedded application can use the same network policy as the one used to bring up the laptop call. However, in the network policy structure, the is_routeable flag should be set to TRUE.

3.9.2.4 DSS_DNS_CONFIG_PARAMS_USE_CACHE_RESULTS

This option is used to specify whether to cache results. By default, this parameter is set to TRUE. If the application does not want to use the cache results, it must set this parameter to FALSE. The data type of param_val_ptr should be boolean and the param_len argument should be sizeof(boolean).

3.9.2.5 DSS_DNS_CONFIG_PARAMS_ADD_TO_CACHE

This option is used to specify whether the results obtained from querying the DNS server should be added to the cache. By default, this parameter is set to TRUE. If the application does not want to add the query results to the cache, it must set this parameter to FALSE. The data type of param_val_ptr should be boolean and the param_len argument should be sizeof(boolean).

Example

```
int16                           dss_errno = 0;
int32                           cb_user_data = 0;
dss_dns_session_mgr_handle_type session_handle;
boolean                         add_to_cache = FALSE;
int16                           retval;

/*-------------------------------------------------------------------
  We create a session with cback_fcn callback function.
-------------------------------------------------------------------*/
session_handle = dss_dns_create_session( cback_fcn,
                                         (void *) &cb_user_data,
                                         &dss_errno );

if( DSS_DNS_SESSION_INVALID_HANDLE == session_handle )
{
  MSG_ERROR("Session could not be created error %d", dss_errno, 0, 0);
}

/*-------------------------------------------------------------------
  Set the ADD_TO_CACHE config parameter for this session.
-------------------------------------------------------------------*/
retval = dss_dns_set_config_params( session_handle,
                                    DSS_DNS_CONFIG_PARAMS_ADD_TO_CACHE,
                                    (void *) &add_to_cache,
                                    sizeof(boolean),
                                    &dss_errno );
if( DSS_ERROR == retval )
{
  MSG_ERROR("Set config param returned error %d", dss_errno);
}
```

3.9.3 dss_dns_get_config_params()

This function is used to get various configuration parameters that are set for a given session.

Parameters

```
int16 dss_dns_get_config_params(
                dss_dns_session_mgr_handle_type   session_handle,
                dss_dns_config_params_enum_type   param_name,
                void                              *param_val_ptr,
                uint32                            param_len,
                int16                             *dss_errno
                )
```

→	dss_dns_get_config_params		
	→	session_handle	Session control block identifier

	→	param_name	Type of parameter to be set. This is an enum of type:
			`typedef enum dss_dns_config_params_enum_type`
			`{`
			`DSS_DNS_CONFIG_PARAMS_MIN,`
			`DSS_DNS_CONFIG_PARAMS_IFACE_ID,`
			`DSS_DNS_CONFIG_PARAMS_USE_CACHE_RESULTS,`
			`DSS_DNS_CONFIG_PARAMS_ADD_TO_CACHE,`
			`DSS_DNS_CONFIG_PARAMS_MAX`
			`} dss_dns_config_params_enum_type;`

	←	param_val_ptr	Pointer to buffer where the parameter needs to be read
	→	param_len	Length of the parameter. The caller specifies the length of the param_val_ptr argument that should be read.
	←	dss_errno	Error code returned to the application in case of an error. Possible values: ▪ DS_EBADF – Invalid session handle ▪ DS_EFAULT – Invalid arguments

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error code value is set in dss_errno

Description

The application calls this function to get a given configuration parameter (specified by param_name argument) to a given value (specified by param_val_ptr argument). The parameters that can be obtained are the same as those described those in the dss_dns_set_config_params function described in Section 3.9.2.

Example

```
int16                           dss_errno = 0;
int32                           cb_user_data = 0;
dss_dns_session_mgr_handle_type session_handle;
dss_iface_id_type               iface_id;
int16                           retval;

/*-------------------------------------------------------------------
  We create a session with cback_fcn callback function.
-------------------------------------------------------------------*/
session_handle = dss_dns_create_session( cback_fcn,
                                         (void *) &cb_user_data,
                                        &dss_errno );

if( DSS_DNS_SESSION_INVALID_HANDLE == session_handle )
{
  MSG_ERROR("Session could not be created error %d", dss_errno, 0, 0);
}

/*-------------------------------------------------------------------
  Get the iface_id config parameter from the session.
-------------------------------------------------------------------*/
retval = dss_dns_get_config_params( session_handle,
                                    DSS_DNS_CONFIG_PARAMS_IFACE_ID,
                                    (void *) &iface_id,
                                    sizeof(dss_iface_id_type),
                                    &dss_errno );
if( DSS_ERROR == retval )
{
  MSG_ERROR("Set config param returned error %d", dss_errno);
}
```

3.9.4 dss_dns_delete_session()

This function deletes a session that was created using dss_dns_create_session.

Parameters

```
int16 dss_dns_delete_session(
                        dss_dns_session_mgr_handle_type   session_handle
                        int16     *dss_errno
)
```

→	dss_dns_delete_session	
	→ session_handle	Session control block identifier
	← dss_errno	Errno returned to the application; possible errno value: ▪ DS_EBADF – Invalid session handle specified

Return value

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in errno

Description

This function should be called after all the query operations associated with a session are complete. This function deletes the session and frees the session control block associated with the session. If the application calls dss_dns_delete_session when there are outstanding DNS queries, all the DNS queries are silently aborted without notifying the application.

Example

```
int16                           dss_errno = 0;
int32                           cb_user_data = 0;
dss_dns_session_mgr_handle_type  session_handle;

/*-------------------------------------------------------------------------
  We create a session with cback_fcn callback function.
-------------------------------------------------------------------------*/
session_handle = dss_dns_create_session( cback_fcn,
                                         (void *) &cb_user_data,
                                         &dss_errno );

if( DSS_DNS_SESSION_INVALID_HANDLE == session_handle )
{
  MSG_ERROR("Session could not be created error %d", dss_errno, 0, 0);
}
/*-------------------------------------------------------------------------
  Delete this session.
-------------------------------------------------------------------------*/
retval = dss_dns_delete_session( session_handle,
                                 &dss_errno );
if( DSS_ERROR == retval )
{
  MSG_ERROR("Session %d could not be deleted", session_handle);
}
```

3.9.5 dss_dns_get_addrinfo()

This function provides a protocol-independent translation of a host name to an address.

Parameters

```
dss_dns_query_handle_type dss_dns_get_addrinfo(
                dss_dns_session_mgr_handle_type   session_handle,
                const char                        *hostname,
                const char                        *service,
                const struct dss_dns_addrinfo     *hints,
                int16                             *dss_errno
)
```

→	dss_dns_get_addrinfo		
	→	session_handle	Pointer identifying the session control block
	→	hostname	String with the domain name of an IPv4 or IPv6 host or dotted-decimal IPv4 or colon-separated/dotted-decimal IPv6 host
	→	service	NULL or decimal port number string
	→	hints	A constant dss_dns_addrinfo structure that directs the use of this function by parameters, including flags, family, sock_type, and protocol
	←	dss_errno	Errno returned back to the application; valid errno values returned are: ▪ DS_EHOSTNOTFOUND – Domain name is not known ▪ DS_ENORECOVERY – Irrecoverable server error occurred ▪ DS_EFAULT – Invalid parameters passed to the function ▪ DS_EBADF – Invalid session handle. ▪ DS_EAFNOSUPPORT – Invalid value for the address family parameter ▪ DS_EOPNOTSUPP – Invalid value for the flags parameter ▪ DS_ENOMEM – Out of memory in the DNS subsystem ▪ DS_NAMEERR – Domain name is malformed ▪ DS_EWOULDBLOCK – DNS servers being queried would invoke the callback with the answer

Return value

This function returns:

- On normal operation, query identifier would be returned to the application and the error code will contain DS_EWOULDBLOCK; the application should store the query identifier to read the query results using the dss_dns_read_addrinfo function
- DSS_ERROR – Error; error code value is placed in errno

Description

This function queries for dss_dns_addrinfo records from lower layers. If there are no errors, the function returns a query identifier for the application specified query. The function dss_dns_get_addrinfo() validates the arguments and internally interacts with the DNS resolver module to send a query to the DNS server. After receiving the reply from the resolver, it signals the application through a registered callback that a response has been received. The application then calls the dss_dns_read_addrinfo() function to read the set of dss_dns_addrinfo records. The dss_dns_addrinfo records have the following structure:

```
typedef struct dss_dns_addrinfo
{
  uint16   ai_flags;   /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST */
  uint16   ai_family;   /* AF_INET, AF_INET6, AF_UNSPEC            */
  uint16   ai_socktype; /* SOCK_STREAM, SOCK_DGRAM                 */
  uint16   ai_protocol; /* IPPROTO_TCP, IPPROTO_UDP                */
  uint16   ai_addrlen;  /* length of ai_addr                       */
  char    ai_cannonname[DSS_DNS_CANONNAME_MAX_LEN];/* canonical    */
  struct sockaddr_storage ai_sockaddr; /* generic socket address   */
} dss_dns_addrinfo;
```

The hostname and service arguments are either NULL pointers or pointers to NULL-terminated strings. If the hostname argument is non-NULL, it should represent a well-formed domain name or an IPv4 or IPv6 numeric IP address as explained in [**Error! Reference source not found.**]. If either of these are not well-formed domain name representations, DS_NAMEERR will be returned through errno.

The hints parameter directs the behavior of the dss_dns_get_addrinfo function. The caller can supply the following structure members in hints:

- ai_family – Indicates the protocol family that the caller is willing to accept. The supported values are AF_INET, AF_INET6, and AF_UNSPEC. If the address family is AF_INET, the caller is willing to accept only IPv4 sockets. If the address family is AF_INET6, the caller is willing to accept only IPv6 sockets. If the address family is AF_UNSPEC, the caller is willing to accept both IPv4 and IPv6 sockets.
- ai_socktype – Indicates the type of socket that the caller is willing to accept. The supported values are SOCK_STREAM for TCP sockets and SOCK_DGRAM for UDP sockets. When ai_socktype is zero, the caller is willing to accept any type of socket.
- ai_protocol – Indicates which transport protocol is desired. Supported values are IPPROTO_TCP and IPPROTO_UDP. A value of zero indicates that the caller is willing to accept sockets with any type of transport protocol.
- ai_flags – Bitwise OR argument of one or more of the following:
  - DSS_DNS_AI_CANONNAME – If this flag is specified, the function shall attempt to determine the canonical name corresponding to hostname, e.g., if hostname is an alias or shorthand notation for a complete name.

- DSS_DNS_AI_NUMERICHOST – If this flag is specified, the hostname shall represent the numeric host address string. Otherwise, the DS_ENAMEERR error is returned. The DNS server is not queried in this case.
- DSS_DNS_AI_PASSIVE – This flag is considered only if the hostname argument is null. If this flag is specified and the hostname argument is null, the returned address is suitable for binding a socket to accept incoming connections. In this case, if the nodename argument is NULL, then the IP address portion of the socket address structure shall be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If this flag is not specified and nodename is NULL, the returned address information shall be suitable for a call to connect() for a connection-mode protocol or for a call to connect(), sendto(), or sendmsg() for a connectionless protocol. In this case, the IP address portion of the socket address structure shall be set to the local loopback address.
- DSS_DNS_AI_V4MAPPED – This flag is valid only if ai_family equals AF_INET6. If this flag is specified and ai_family is AF_INET6, then dss_dns_get_addrinfo() shall return the IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses. For example, if no AAAA records are found, a query is made for A records and any that are found are returned as IPv4-mapped IPv6 addresses.
- DSS_DNS_AI_ALL – This flag is valid when the DSS_DNS_AI_V4MAPPED option is specified. In this case, the DNS queries will be made for both A and AAAA records. Results of A records will be returned as IPv4-mapped IPv6 records.
- DSS_DNS_AI_ADDRCONFIG – If this flag is specified, then a query for AAAA records will be made only if an IPv6 address is configured for the local system (other than the loopback address), and a query for A records will be made only if an IPv4 address is configured for the local system (other than loopback address).

The callback function information is stored in the session control block. The dss_dns_get_addrinfo function signals the application after the querying is complete. The application reads these records into its own memory by using the API function dss_dns_read_addrinfo().

Example

```
/*-------------------------------------------------------------------
  A DNS session has been created using dss_dns_create_session. Invoke
  DNS resolver using dss_dns_get_nameinfo
-------------------------------------------------------------------*/
int16                          dss_errno = 0;
char                           domain_name[] = “www.qualcomm.com”;
dss_dns_query_handle_type      query_handle;


if( DSS_DNS_SESSION_INVALID_HANDLE == session_handle )
{
  MSG_ERROR(“Session could not be created error %d”, dss_errno,0,0);
}

/*-------------------------------------------------------------------
  Invoke DNS resolver using dss_dns_get_addrinfo
```

```
------------------------------------------------------------------*/
query_handle = dss_dns_get_addrinfo( session_handle,
                                     domain_name,
                                     NULL,
                                     NULL,
                                     &dss_errno );
if( DSS_DNS_QUERY_INVALID_HANDLE == query_handle )
{
  MSG_ERROR("Get_addrinfo returned error %d", dss_errno);
}
/*-----------------------------------------------------------------
  Now wait for the callback (registered during create_session) to be
  called...
------------------------------------------------------------------*/
```

3.9.6 dss_dns_read_addrinfo()

This function reads the records that were queried through dss_dns_get_addrinfo().

Parameters

```
int16 dss_dns_read_addrinfo(
                dss_dns_session_mgr_handle_type   session_handle
                dss_dns_query_handle_type         query_handle,
                dss_dns_addrinfo                  *results,
                uint16                            num_records,
                int16                             *dss_errno
)
```

→	dss_dns_read_addrinfo		
	→	session_handle	Session identifier
	→	query_handle	Query identifier
	←	results	Buffer where the records will be read
	→	num_records	Number of records to be read
	←	dss_errno	Error code in case of error; valid error values are: ▪ DS_EBADF – Indicates an invalid session handle ▪ DS_EWOULDBLOCK – Indicates that the query is still not complete; try again later

Return value

This function returns:

- DSS_SUCCESS – Successful operation; results will be stored in the results argument
- DSS_ERROR – Error; error condition value is placed in errno

Description

This function allows the application to read the dss_dns_addrinfo records. The application provides memory for the records to be read. The application can also specify how many records it wants to read through this function.

After a call to the dss_dns_read_addrinfo() function, all the records corresponding to the query and other internal data structures used for performing the query are freed. Hence, a call to dss_dns_read_addrinfo() again will return DSS_ERROR and DS_EBADF in dss_errno since the query handle would no longer be valid.

Example

```
/*-------------------------------------------------------------------
  dss_dns_get_addrinfo invokes the registered callback for the
  session
-------------------------------------------------------------------*/
void cback_fcn
(
  dss_dns_session_mgr_handle_type   session_handle,
  dss_dns_query_handle_type         query_handle,
  dss_dns_api_type_enum_type        api_type,
  uint16                            num_records,
  void                            * user_data_ptr,
  int16                             dss_errno
)
{
  int16                             retval;
  int16                             errno;
/*-------------------------------------------------------------------*/
  static dss_dns_addrinfo result_buf[MAX_RECORDS];

  retval = dss_dns_read_addrinfo( session_handle,
                                  query_handle,
                                  result_buf,
                                  (num_records > MAX_RECORDS)?
                                  MAX_RECORDS : num_records,
                                  &errno );
  if( DSS_ERROR == retval )
  {
    MSG_ERROR("dss_dns_read_addrinfo returned error %d", errno);
  }
  else
  {
    /* Process result_buf... */
  }
}
```

3.9.7 dss_dns_get_nameinfo()

This function maps a socket address to a corresponding node name and service location.

Parameters

```
dss_dns_query_handle_type dss_dns_get_nameinfo(
                      dss_dns_session_mgr_handle_type   session_handle
                      const struct sockaddr             *sa_ptr,
                      uint16                            sa_len,
                      uint32                            flags,
                      int16                             *dss_errno
)
```

→	dss_dns_get_nameinfo		
	→	session_handle	Session identifier
	→	sa_ptr	Pointer to a generic socket address
	→	sa_len	Length of sockaddr; determines if the sockaddr is of type sockaddr_in or sockaddr_in6
	→	flags	Flags that direct the operation of get_nameinfo; can be the OR of one or more of the following: ▪ DSS_DNS_NI_FLAGS_NOFQDN ▪ DSS_DNS_NI_FLAGS_NUMERICHOST
	←	dss_errno	Errno returned to the application; possible error values are: ▪ DS_EFAULT – Indicates faulty parameters ▪ DS_EBADF – Indicates an invalid session handle ▪ DS_ENOMEM – Indicates a memory allocation failure ▪ DS_ENAMEERR – Indicates that the domain name is malformed ▪ DS_EWOULDBLOCK – Indicates that the operation would block ▪ DS_EOPNOTSUPP – Indicates that the operation is not supported ▪ DS_EAFNOTSUPP – Indicates that the address family is not supported ▪ DS_ESYSTEM – Indicates that some irrecoverable system error occurred

Return value

This function returns:

- On normal operation, the query identifier would be returned to the application and the error code will contain DS_EWOULDBLOCK. The application should store the query identifier to read the query results using the dss_dns_read_nameinfo function.
- On error, DSS_ERROR would be returned and errno will contain the error code.

Description

The dss_dns_get_nameinfo function converts a socket address structure to a pair of hostname and service name parameters. The dss_dns_get_nameinfo() function validates the arguments and internally interacts with the DNS resolver module to send the query to the DNS server. After receiving the results from the resolver, it signals the application using the application registered callback for the session. The application uses the dss_dns_read_nameinfo API function to read the results.

The argument sa_ptr points to a socket structure to be translated. The sa_len argument holds the size of the socket address structure. The flags argument directs the translation. The flags argument can be an OR of one of more of the following.

- DSS_DNS_NI_FLAGS_NOFQDN – If this flag is set, only the nodename portion of the fully-qualified domain name (FQDN) is returned.
- DSS_DNS_NI_FLAGS_NUMERICHOST – If this flag is set, no DNS lookup is performed; the numeric host address representation is returned to the application.

Example

```
/*-------------------------------------------------------------------
  A DNS session has been created using dss_dns_create_session. Invoke
  DNS resolver using dss_dns_get_nameinfo
-------------------------------------------------------------------*/
dss_dns_query_handle_type    query_handle;
const struct sockaddr_in     sock_in;
int16                        dss_errno;

memset( &sock_in, 0, sizeof(struct sockaddr_in) );
sock_in.sin_family = AF_INET;
sock_in.sin_port = 0;
(void) dss_inet_aton( “199.106.114.68”, &(sock_in.sin_addr) );

query_handle = dss_dns_get_nameinfo( session_handle,
                                     (struct sockaddr *) &sock_in,
                                     sizeof(struct sockaddr_in),
                                     0,
                                     &dss_errno );
if( DSS_DNS_QUERY_INVALID_HANDLE == query_handle )
{
  MSG_ERROR(“dss_dns_get_nameinfo returned error %d”, dss_errno);
}
/*-------------------------------------------------------------------
  Now wait for the callback (registered during create_session) to be
  called...
-------------------------------------------------------------------*/
```

3.9.8 dss_dns_read_nameinfo()

This function reads the records that were queried through dss_dns_get_nameinfo().

Parameters

```
int16 dss_dns_read_nameinfo(
   dss_dns_session_mgr_handle_type       session_handle
   dss_dns_query_handle_type             query_handle,
   dss_dns_nameinfo                      *results,
   uint16                                num_records,
   int16                                 *dss_errno
)
```

→	dss_dns_read_nameinfo		
	→	session_handle	Session identifier
	→	query_handle	Query identifier

	←	results	Buffer where the records will be read; it is of the following format:
			`typedef struct dss_dns_nameinfo`
			`{`
			`  char  hostname[DSS_DNS_MAX_DOMAIN_NAME_LEN];`
			`  char  servname[DSS_DNS_MAX_SERV_NAME_LEN];`
			`} dss_dns_nameinfo;`

	→	num_records	Number of records that the application wants to read
	←	dss_errno	Error code in case of error; possible values are: ▪ DS_EBADF – Indicates an invalid session or query handle ▪ DS_EWOULDBLOCK – Indicates that the query operation is not yet complete

Return value

This function returns:

- DSS_SUCCESS – Successful operation; results will be stored in results argument.
- DSS_ERROR – Error; error condition value is placed in errno

Description

This function reads the dss_dns_nameinfo records that were returned as a result of the DNS query invoked through dss_dns_get_nameinfo. It should be called when the callback registered with the session is invoked. The buffer space for copying the records is provided by the application.

This function is similar to dss_dns_read_addrinfo() in its operation. It frees the resource records and internal data structures associated with the query after a call to dss_dns_read_nameinfo().

Example

```
/*-------------------------------------------------------------------------
  dss_dns_get_nameinfo invokes the registered callback for the session
-------------------------------------------------------------------------*/
void cback_fcn
(
  dss_dns_session_mgr_handle_type   session_handle,
  dss_dns_query_handle_type         query_handle,
  dss_dns_api_type_enum_type        api_type,
  uint16                            num_records,
  void                            * user_data_ptr,
  int16                             dss_errno
)
{
  int16                             retval;
  int16                             errno;
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
  static dss_dns_nameinfo result_buf[MAX_RECORDS];

  retval = dss_dns_read_nameinfo( session_handle,
                                  query_handle,
                                  result_buf,
                                  (num_records > MAX_RECORDS)?
                                  MAX_RECORDS : num_records,
                                  &errno );
  if( DSS_ERROR == retval )
  {
    MSG_ERROR("dss_dns_read_nameinfo returned error %d", errno);
  }
  else
  {
    /* Process result_buf... */
  }
}
```

3.9.9 dss_getipnodebyname()

This function maps a domain name to its associated IP addresses.

Parameters

```
struct dss_hostent * dss_getipnodebyname (
                                          char             *name,
                                          int32             af,
                                          int32             flags,
                                          dss_dns_cb_f_type callback
                                           void            *cb_voidptr,
                                           int32           *dss_errno
                                          )
```

→	dss_getipnodebyname		

	→	name	Domain name string to be resolved, i.e., www.qualcomm.com; the answer is returned in struct dss_hostent that is defined as:
			
			`struct dss_hostent`
			`{`
			`  /* Official host name */`
			`  char  *h_name;`
			`  /* NULL-terminated alias list */`
			`  char **h_aliases;`
			`  /* host address type */`
			`  int    h_addrtype;`
			`  /* length of address */`
			`  int    h_length;`
			`  /* NULL-terminated address list */`
			`  char **h_addr_list;`
			`};`

	→	af	Address family of the IP address requested; currently only AF_INET and AF_INET6 are supported
	→	flags	Options field; currently, no option is supported and this parameter must be zero

	→	callback	Callback function; if the domain name is not found and the operation would block, then the answer is returned in the callback; the callback function prototype is:
			
			`typedef void (*dss_dns_cb_f_type)`
			`(`
			`  struct dss_hostent *phostent,`
			`  void                        *cb_voidptr,`
			`  int32                        dss_errno`
			`);`

	→	cb_voidptr	Cookie registered with the DNS resolver to be returned with the callback

	←	dss_errno	Error code, if any, that occurred; error code values are: ▪ DS_EHOSTNOTFOUND – Indicates that the requested domain name was not found ▪ DS_EFAULT – Indicates that invalid parameters were passed to the function ▪ DS_ENORECOVERY – Indicates an irrecoverable server error occurred ▪ DS_EOPNOSUPP – Indicates that invalid flag values were passed ▪ DS_ETRYAGAIN – Indicates a temporary and transient error, i.e., resolver out of resources, server temporarily unreachable; try again later ▪ DS_EWOULDBLOCK – Indicates that the operation would block, and the answer will be returned in the callback ▪ DS_ENOMEM – Indicates that the DNS subsystem is out of memory ▪ DS_NAMEERR – Indicates that the domain name is malformed ▪ DS_EAFNOSUPPORT – Indicates that the address family specified is not supported

Return value

This function returns:

- NULL – Indicates that the domain name could not be resolved; ff dss_errno is set to DS_EWOULDBLOCK then the answer is returned later in the callback
- Pointer to dss_hostent structure – Returned if the mapping is successful and an answer is found in the DNS cache; the memory associated with the dss_hostent structure must be freed by a call to dss_freehostent().

Description

The application calls this function to map a domain name string to its associated IP addresses. If a mapping is found in the DNS cache, the function returns it in a dss_hostent structure. A NULL is returned with the dss_errno parameter set to DS_EWOULDBLOCK if the mapping has to be acquired. In this case, the answer is returned later in the callback function. The memory associated with the dss_hostent structure must be freed by a call to dss_freehostent(). This function is similar to the one proposed in [**Error! Reference source not found.**] for domain name lookup.

Example

```
char               *name       = ”www.qualcomm.com”;
int32               af         = AF_INET;
int32               flags      = 0;
int32               dss_errno  = 0;
int32               cb_voidptr = 0;
struct dss_hostent *phostent   = NULL;

/*-------------------------------------------------------------------
  To resolve www.qualcomm.com to an IPv4 address with a cookie
  (cb_voidptr)value of 0, the flags field needs to be 0.
  callback_fcn_ptr is the pointer to the callback function.
-------------------------------------------------------------------*/
phostent = dss_getipnodebyname( name,
                                af,
                                flags,
                                callback_fcn_ptr,
                                (void *) cb_voidptr,
                               &dss_errno
                              );

if(phostent == NULL)
{
  MSG_LOW(“The dss_hostent will be returned in the callback.”,0,0,0);
}
```

3.9.10 dss_getipnodebyaddr()

This function maps an IP address to its associated domain names.

Parameters

```
struct dss_hostent * dss_getipnodebyaddr (
                                    void             *addr,
                                    int32             len,
                                    int32             af,
                                    dss_dns_cb_f_type callback
                                    void             *cb_voidptr,
                                    int32            *dss_errno
                                    )
```

→	dss_getipnodebyaddr		

	→	addr	IP address to be mapped, i.e., 0x812e0d26 for a 32-bit IPv4 address; the answer is returned in struct dss_hostent that is defined as:
			`struct dss_hostent`
			`{`
			`  /* Official host name */`
			`  char  *h_name;`
			`  /* NULL-terminated alias list */`
			`  char **h_aliases;`
			`  /* host address type */`
			`  int    h_addrtype;`
			`  /* length of address */`
			`  int    h_length;`
			`  /* NULL-terminated address list */`
			`  char **h_addr_list;`
			`};`

	→	len	Length of the addr field
	→	af	Address family of the IP address passed; currently, only AF_INET and AF_INET6 are supported

	→	callback	Callback function; if the domain name is not found in the DNS cache and the operation would block, then the answer is returned in the callback; the callback function prototype is:
			`typedef void (*dss_dns_cb_f_type)`
			`(`
			`  struct dss_hostent *phostent,`
			`  void               *cb_voidptr,`
			`  int32               dss_errno`
			`);`

	→	cb_voidptr	Cookie registered with the DNS resolver to be returned with the callback

	←	dss_errno	Error code, if any, that occurred; possible error code values are: ▪ DS_EFAULT – Indicates that invalid parameters were passed to the function ▪ DS_EHOSTNOTFOUND – Indicates that the requested domain name was not found ▪ DS_ENORECOVERY – Indicates that an irrecoverable server error occurred ▪ DS_ETRYAGAIN – Indicates a temporary or transient error, i.e., resolver out of resources, server temporarily unreachable; try again later ▪ DS_EWOULDBLOCK – Indicates that the operation would block, the answer will be returned in the callback ▪ DS_ENOMEM – Indicates that the DNS subsystem is out of memory ▪ DS_NAMEERR – Indicates that the query is malformed ▪ DS_EAFNOSUPPORT – Indicates that the specified address family is not supported

Return value

This function returns:

- NULL – Indicates that the domain name could not be resolved; if dss_errno is set to DS_EWOULDBLOCK, the answer is returned later in the callback
- Pointer to dss_hostent structure – Returns a struct dss_hostent if the mapping is found in the DNS cache; the memory associated with the dss_hostent structure must be freed by a call to dss_freehostent()

Description

The application calls this function to map an IP address to its associated domain names. If a mapping is found in the DNS cache, the function returns it in a dss_hostent structure. A NULL is returned with the dss_errno parameter set to DS_EWOULDBLOCK if the mapping has to be acquired. In this case, the answer is returned later in the callback function. The memory associated with the dss_hostent structure must be freed by a call to dss_freehostent(). This function is similar to the one proposed in [**Error! Reference source not found.**] for reverse domain name lookup.

Example

```
struct in_addr      addr       = {0x0a2e0d26};
int32               af         = AF_INET;
int32               len        = sizeof( addr );
int32               dss_errno  = 0;
int32               cb_voidptr = 0;
struct dss_hostent *phostent   = NULL;
/*-------------------------------------------------------------------
  To resolve the IPv4 address 0x0a2e0d26 with a cookie (cb_voidptr)
  value of 0. callback_fcn_ptr is the pointer to the callback
  function.
  -------------------------------------------------------------------*/
```

```
phostent = dss_getipnodebyaddr( (void *) addr,
                                 len,
                                 af,
                                 callback_fcn_ptr,
                                 (void *) cb_voidptr,
                                &dss_errno
                               );

if(phostent == NULL)
{
  MSG_LOW("The dss_hostent will be returned in the callback.",0,0,0);
}
```

3.9.11 dss_freehostent()

This function frees the dss_hostent structure returned by dss_getipnodebyname() or dss_getipnodebyaddr().

Parameters

```
void dss_freehostent(
                      struct dss_hostent *phostent
                    )
```

→	dss_freehostent		
	→	phostent	Pointer to the dss_hostent structure to be freed

Description

This function is called with a pointer to a dss_hostent structure. The dss_hostent must have been returned earlier by dss_getipnodebyname() or dss_getipnodebyaddr(). This function is similar to the one proposed in [**Error! Reference source not found.**] for freeing memory returned by domain name lookup functions.

Example

```
struct dss_hostent *phostent = NULL;

phostent = call to dss_getipnodebyname() or dss_getipnodebyaddr()

/*-----------------------------------------------------------------
  The non-NULL dss_hostent structure pointer phostent is returned by
  a call to dss_getipnodebyname() or dss_getipnodebyaddr()
  -----------------------------------------------------------------*/
```

```
dss_freehostent( phostent );
```

3.10 Ping

3.10.1 dss_ping_init_options()

This function initializes the ping options structure that is to be passed to the dss_ping_start() function.

Parameters

```
void dss_ping_init_options (
              dss_ping_config_type  *ping_configs
              )
```

→	dss_ping_init_options		

	←	*ping_configs	Pointer to ping options structure.
			The following structure is used to define the ping options:
			`typedef struct`
			`{`
			`  uint32 num_data_bytes;`
			`  uint32 ping_interval_time;`
			`  uint32 ping_response_time_out;`
			`  int    cookie;`
			`uint8  num_pings;`
			`  uint8  ttl;`
			`} dss_ping_config_type;`
			
			The num_data_bytes field in the dss_ping_config_type structure specifies the number of bytes that the ping packet should carry in its payload. The default value is 64 bytes.
			The ping_interval_time field in the dss_ping_config_type structure specifies the interval between each ping that is sent out. Value is specified in milliseconds. The default value is 2000 ms.
			The ping_response_time_out field in the dss_ping_config_type structure specifies the timeout value for each ping request. Any ping response received after the timeout shall not be processed. Value specified in milliseconds. The default value is 10000 ms.
			The cookie field in the dss_ping_config_type structure is an internal field used by the ping API.
			The num_pings field in the dss_ping_config_type structure specifies the number of times the ping request is to be sent. The default value is four.
			The ttl field in the dss_ping_config_type structure specifies the Time-To-Live (TTL) value for the IP packet over which the ping (ICMP ECHO Request message) is sent. The default value is 255.

Return value

This function returns:

- DSS_SUCCESS – Success
- DSS_ERROR – Error; this will happen if a NULL pointer is passed as the function parameter

Dependencies

This function must be invoked before calling the dss_ping_start() function.

Description

This function initializes the ping options data structure that is passed as the function parameter. An application must call this function before supplying the ping options to the dss_ping_start() function. Once the ping options structure has been initialized, the application can modify various fields.

Example

```
dss_ping_config_type  my_ping_options;
int16  errno;

/*-------------------------------------------------------------------
Initialize the ping options and then modify it, if required. Use this
initialized ping options structure when calling the dss_ping_start() to
ping a destination.
-------------------------------------------------------------------*/
dss_ping_init_options(&my_ping_options);

/* Now modify options, if desired */
my_ping_options.num_pings = 100; /* Ping 100 times instead of default value
*/

dss_ping_start( some_net_policy_ptr,
                some_dest_addr_ptr,
                some_dest_addr_type,
                &my_ping_options,
                app_cb_fn,
                app_summary_cb_fn,
                app_user_data,
                &errno );
```

3.10.2 dss_ping_start()

This function pings the specified destination with the indicated ping options.

Parameters

```
dss_ping_handle dss_ping_start        (
     dss_net_policy_info_type             *net_policy
     char                                 *dest_addr
     dss_ping_ip_addr_enum_type           dest_addr_type
     struct dss_ping_config_type          *app_ping_options
     dss_ping_callback_fn_type            app_callback_fn
     dss_ping_sess_summary_callback_fn_   app_sess_summary_callback_fn
    type
     void                                 *app_user_data
     int16                                *dss_errno
     )
```

→	dss_ping_start		
	→	*net_policy	Pointer to policy information; the function uses this policy information to bring up a particular network interface in order to ping the specified destination; the application can specify a NULL pointer to indicate that the default policy can be used; otherwise, if the application wants to specify a particular policy, it must initialize the policy information by calling dss_init_net_policy_info()
	→	*dest_addr	Pointer to destination address; this is the destination that is to be pinged; it can be an IP address (IPv4 or IPv6) or a domain name
	→	dest_addr_type	Address family type (IPv4 or IPv6) of the destination address
	→	*app_ping_options	Pointer to ping options structure

	→	app_callback_fn	Pointer to application's callback function; this function will be invoked whenever a ping response arrives or a ping timeout occurs, i.e., no ping response was received within a certain time; app_callback_fn is of type dss_ping_callback_fn_type, and is defined as:
			`void (*dss_ping_callback_fn_type)`
			`(`
			`  dss_ping_handle  ping handle,`
			`  dss_ping_stats_type  *ping_stats,`
			`  void  *user_data,`
			`  int16  errno`
			`)`
			
			▪ ping_handle – Indicates the ping session handle returned by dss_ping_start()
			▪ ping_stats – Indicates the statistics for this single ping request, e.g., round trip time, resolved IP address, etc.; the following structure is used to convey information about the ping statistics:
			
			`struct dss_ping_stats_type`
			`{`
			`  uint32  rtt;`
			`  uint16  icmp_seq_num;`
			`  char  resolved_ip_addr[64];`
			`};`
			▫ rtt – Round trip time for the ping
			▫ icmp_seq_num – Sequence number of the ping packet
			▫ resolved_ip_addr – Indicates the resolved IP address if a domain name is specified
			▪ user_data – Indicates the user data that was specified by application in dss_ping_start()
			▪ errno – Indicates the response receive success (DSS_SUCCESS) or response timeout (DS_ETIMEDOUT)

	→	app_sess_summary_ callback_fn	Pointer to application's session summary callback function; this function will be invoked only once when a ping session ends; a ping session can end due to:
			▪ User's request
			▪ Some error on the API side (network down, memory unavailable, etc.)
			▪ On graceful end, i.e., all ping requests were sent and their responses were accounted for (either timed-out or received)
			app_sess_summary_callback_fn is of type dss_ping_sess_summary_callback_fn_type, and is defined as:
			`void`
			`(*dss_ping_sess_summary_callback_fn_type)`
			`(`
			`dss_ping_handle ping handle,`
			`dss_ping_session_stats_type`
			`*ping_session_stats,`
			`void *user_data,`
			`dss_ping_session_close_reason_type`
			`reason,`
			`int16 errno`
			`)`
			▪ ping_handle – Indicates the handle returned by dss_ping_start()
			▪ ping_session_stats – Indicates the statistics for the entire ping session, e.g., average round trip time, maximum round trip time etc.; the following structure is used to convey information about the ping session statistics:
			`struct dss_ping_session_stats_type`
			`{`
			`uint32 min_rtt;`
			`uint32 max_rtt;`
			`uint32 avg_rtt;`
			`uint8 num_pkts_sent;`
			`uint8 num_pkts_recvd;`
			`uint8 num_pkts_lost;`
			`};`
			▫ min_rtt – Indicates the minimum RTT
			▫ max_rtt – Indicates the maximum RTT
			▫ avg_rtt – Indicates the average RTT
			▫ num_pkts_sent – Indicates the number of ping requests that were sent out
			▫ num_pkts_recvd – Indicates the number of ping responses that were received
			▫ num_pkts_lost – Indicates the number of ping requests for which no response was received
			▪ user_data – Indicates the user data that was specified by the application in dss_ping_start()

			▪ reason – Indicates the reason why the ping session ended; reasons are defined as follows:
			
			`enum dss_ping_session_close_reason_type`
			`{`
			`  DSS_PING_USER_REQ,`
			`  DSS_PING_SESSION_COMPLETE,`
			`  DSS_PING_ERROR,`
			`};`
			
			▫ DSS_PING_USER_REQ – Indicates that the user requested close
			▫ DSS_PING_SESSION_COMPLETE – Indicates a normal close (all requests sent and responses accounted for)
			▫ DSS_PING_ERROR – Indicates that the session closed because of some internal error; the error condition is placed in the errno field
			▪ errno – Indicates if the reason is indicated as DSS_PING_ERROR; this field conveys the error condition

	→	`*app_user_data`	▪ Pointer to application's user data; this value will be used when invoking the application's ping callback functions
	←	`*errno`	▪ Pointer to the returned error value that is passed by reference; possible values are: ▪ DS_EADDRREQ – Indicates that the destination address was not specified ▪ DS_EFAULT – Indicates that an invalid parameter was specified, e.g., NULL pointer for callback function ▪ DS_EMSGSIZE – Indicates that the requested ping data byte size is too large ▪ DS_ENOMEM – Indicates that memory was unavailable ▪ DS_NAMEERR – Indicates that the destination name was malformed ▪ DS_ETRYAGAIN – Indicates temporary and transient errors, e.g., DNS resolver out of resources ▪ DS_EHOSTNOTFOUND – Indicates that the domain name is not known

Return value

Note that the function is nonblocking so it returns after validating the parameters and starting a ping session. Any network-related errors would be indicated through the callback function.

This function returns:

- On success, returns a ping handle
- On error, returns DSS_ERROR and places the error condition value in *errno

Dependencies

The application must initialize the app_ping_options being passed to this function by calling dss_ping_init_options().

Additionally, if the application wants to specify a particular network policy for the ping session, it must initialize the net_policy being passed to dss_ping_start() by calling dss_init_net_policy_info(). Otherwise, to request dss_ping_start() to use the default network policy, net_policy can be set to NULL.

Description

This function is used to ping some destination address. It is nonblocking and will return immediately with a ping handle to the ping session. The statistics for each ping will be indicated through a particular application callback function (app_callback_fn), and the statistics for the entire session will be indicated through another application callback function (app_sess_summary_callback_fn).

Currently, the DS protocol stack supports only policy-based routing (regardless of the IP address and the IP address family). So, for pinging, the dss_ping_start() API will use the interface specified through the net_policy parameter. If the user has not specified any specific interface, the API will use a default interface.

3.10.3 dss_ping_stop()

This function aborts the specified ping session.

Parameters

```
int16 dss_ping_stop (
                dss_ping_handle    ping_handle
                )
```

→	dss_ping_stop		
	→	ping_handle	Handle to the ping session that is to be stopped

Return value

This function returns:

- DSS_SUCCESS – Indicates that the specified ping session was found, i.e., ping handle is valid
- DSS_ERROR – Indicates that the specified ping session was not found, i.e., ping handle is invalid

Dependencies

None

Description

This function is used to stop an ongoing ping session that is identified through the ping handle. dss_ping_stop() will tear down any interface that was brought up for the ping session.

3.11 Utility functions

3.11.1 dss_inet_aton()

This function converts an IPv4 address from standard dotted quad notation (where each dot-separated value can be hex, octal, or decimal) to a 4-byte host address form.

Parameters

```
int32 dss_inet_aton(
                      const char     *cp,
                      struct in_addr *addr
                      )
```

→	dss_inet_aton		
	→	cp	IPv4 address in dotted string form, i.e., 10.46.30.120

	←	addr	The converted 4-byte host address struct in_addr is defined as:
			
			`struct in_addr{`
			`  uint32 s_addr;`
			`};`

Return value

This function returns:

- DSS_SUCCESS – Converted address successfully
- DSS_ERROR – Some error occurred

Description

This function is called with the IPv4 address in dotted string form. The address is converted to the 4-byte host address form.

Example

```
struct in_addr      addr       = {0};
char               *addr_str   = “10.46.30.120”;
int32               dss_errno;
/*-------------------------------------------------------------------
  To convert the IPv4 address string “10.46.30.120” to the four byte
  host address form in network byte order.
  -------------------------------------------------------------------*/
dss_errno = dss_inet_aton( addr_str, &addr );

if(dss_errno == DSS_SUCCESS)
{
  MSG_MED(“dss_inet_aton() returned address: 0x%x”, addr.s_addr,0,0);
}
else
{
  MSG_HIGH( “dss_inet_aton() returned error”, 0, 0, 0 );
}
```

3.11.2 dss_inet_ntoa()

This function converts an IPv4 address from 4-byte host address form to standard dotted decimal form.

Parameters

```
int32 dss_inet_ntoa(
                      struct in_addr *addr,
                      uint8     *buf,
                      int32      buflen
                     )
```

→	dss_inet_ntoa		

	→	addr	IPv4 address in 4-byte host address form
			struct in_addr is defined as:
			
			`struct in_addr{`
			`  uint32 s_addr;`
			`};`

	←	buf	Buffer for IPv4 address in converted dotted string form
	→	buflen	Buffer length

Return value

This function returns:

- DSS_SUCCESS – Converted address successfully
- DSS_ERROR – Some error occurred

Description

This function is called with the IPv4 address in 4-byte host address form. The address is converted to the dotted-string form.

Example

```
#define BUFLEN 20

struct in_addr      addr       = {0x0a2e1e78};
char                buf[BUFLEN];
int32               buflen     = BUFLEN;
/*-------------------------------------------------------------------
  To convert the IPv4 address 0x0a2e1e78 from the four byte host
  address form to the dotted string form “10.46.30.120”
  -------------------------------------------------------------------*/
dss_errno = dss_inet_aton( &addr, buf, buflen );

if(dss_errno == DSS_SUCCESS)
{
 MSG_MED( “dss_inet_ntoa() returned address: %s”, buf, 0, 0 );
}
else
{
  MSG_HIGH( “dss_inet_ntoa() returned error”, 0, 0, 0 );
}
```

3.11.3 dss_inet_pton()

This function converts an IP address in standard textual presentation format (i.e., dotted decimal if it is an IPv4 address or colon-separated, 16-bit address pieces if it is an IPv6 address) to its standard numerical representation in network byte order. The function conforms to the definition of inet_pton() in [**Error! Reference source not found.**].

Parameters

```
int32 dss_inet_pton(
                      const char    *src,
                      int32          af,
                      void          *dst,
                      uint32         dst_size,
                      int16         *dss_errno
                      )
```

→	dss_inet_pton		
	→	src	IP address (v4 or v6) in presentation form (dotted decimal for IPv4 and colon-separated for IPv6)
	→	af	Address family of the address in src argument. AF_INET for an IPv4 address and AF_INET6 for an IPv6 address
	←	dst	Memory for the converted IP address in numerical form
	→	dst_size	Size of the memory passed in the dst argument. Should be at least sizeof(struct in_addr) bytes of memory for an IPv4 and sizeof(struct in6_addr) bytes for an IPv6 address
	←	dss_errno	Error code in case of a return of DSS_ERROR – this argument is not set in case of a successful return

Return value

This function returns:

- DSS_SUCCESS – The function converted the address successfully.
- DSS_ERROR – Some error occurred. The error code is returned in the dss_errno argument and can be one of the following:
  - DS_EFAULT – Invalid arguments passed to function
  - DS_EAFNOSUPPORT– Invalid value for the address family argument
  - DS_NAMEERR – Malformed address passed in src argument
  - DS_EMSGTRUNC– Insufficient buffer space in dst argument

Description

This function is called with an IP (either v4 or v6) address in textual presentation format which it converts into the standard numerical format in network byte order. The function accepts only dotted decimal representation of an IPv4 address; it does not accept hexadecimal or octal representations, or fewer than four address octets. The function behavior is consistent with [**Error! Reference source not found.**], which describes socket extensions for IPv6.

Example

```
#include “dssdns.h”

char               *src      = “10.46.30.120”;
struct in_addr      dst      = {0};
int32               af       = AF_INET;
uint32              dst_size = sizeof(struct in_addr);
int16               dss_errno;
int32               ret;

/*-------------------------------------------------------------------
  To convert the IPv4 address string “10.46.30.120” to the four byte
  numerical form in network byte order.
-------------------------------------------------------------------*/
ret = dss_inet_pton( src, af, &dst, dst_size, &dss_errno );

if(ret == DSS_SUCCESS)
{
  MSG_MED( “dss_inet_pton() returned address: 0x%x”, addr.s_addr, 0, 0 );
}
else
{
  MSG_HIGH( “dss_inet_pton() returned error %d”, dss_errno, 0, 0 );
}
```

Example

```
#include “dssdns.h”

char             *src     = “2001:DB8::8:800:200C:417A”;
struct in6_addr     dst      = {0};
int32               af       = AF_INET6;
uint32              dst_size = sizeof(struct in6_addr);
int16               dss_errno;
int32               ret;

/*-------------------------------------------------------------------
  To convert the IPv6 address string “2001:DB8::8:800:200C:417A” to
  the sixteen byte numerical address form in network byte order.
-------------------------------------------------------------------*/
ret = dss_inet_pton( src, af, &dst, dst_size, &dss_errno );

if(ret == DSS_SUCCESS)
{
```

```
        MSG_MED( “dss_inet_pton() returned success”, 0, 0, 0 );
      }
      else
      {
        MSG_HIGH( “dss_inet_pton() returned error %d”, dss_errno, 0, 0 );
      }
```

3.11.4 dss_inet_ntop()

This function converts an IP (v4 or v6) address in standard numerical format in network byte order to its corresponding textual presentation format (i.e., dotted decimal if it is an IPv4 address or colon-separated, 16-bit address pieces if it is an IPv6 address). The function conforms to the definition of inet_ntop() in [**Error! Reference source not found.**].

Parameters

```
int32 dss_inet_ntop(
                      const char    *src,
                      int32          af,
                      void          *dst,
                      uint32         dst_size,
                      int16         *dss_errno
                      )
```

→	dss_inet_pton		
	→	src	IP address (v4 or v6) in standard numerical format in network byte order
	→	af	Address family of the address in src argument, AF_INET for an IPv4 address and AF_INET6 for an IPv6 address
	←	dst	Memory for the converted IP address in standard textual presentation format
	→	dst_size	Size of the memory passed in the dst argument; should have INET_ADDRSTRLEN bytes of memory for an IPv4 and INET6_ADDRSTRLEN bytes of memory for an IPv6 address; these macros defined in dssdns.h
	←	dss_errno	Error code in case of a return of DSS_ERROR; argument not set in case of a successful return

Return value

This function returns:

- DSS_SUCCESS – The function converted the address successfully
- DSS_ERROR – Some error occurred. The error code is returned in the dss_errno argument and can be one of the following:
  - DS_EFAULT – Invalid arguments passed to function
  - DS_EAFNOSUPPORT – Invalid value for the address family argument
  - DS_NAMEERR – Malformed address passed in src argument
  - DS_EMSGTRUNC – Insufficient buffer space in dst argument
  - DS_ENORECOVERY – Nonrecoverable failure to convert the address

Description

This function is called with an IP (either v4 or v6) address in standard numerical format in network byte order which it converts into the standard textual presentation form. The function

behavior is consistent with [**Error! Reference source not found.**], which describes sockets extensions for IPv6.

Example

```
#include “dssdns.h”

struct in_addr      src       = {0x0a2e1e78};
char                dst[INET_ADDRSTRLEN];
int32               af         = AF_INET;
uint32              dst_size   = INET_ADDRSTRLEN;
int16               dss_errno;
int32               ret;

/*-------------------------------------------------------------------
  To convert the IPv4 address 0x0a2e1e78 from the four byte numerical
  form in network byte order to the dotted string form
  “10.46.30.120”.
-------------------------------------------------------------------*/
ret = dss_inet_ntop( &src, af, dst, dst_size, &dss_errno );

if(ret == DSS_SUCCESS)
{
  MSG_MED( “dss_inet_ntop() returned success”, 0, 0, 0 );
}
else
{
  MSG_HIGH( “dss_inet_ntop() returned error %d”, dss_errno, 0, 0 );
}
```

Example

```
#include “dssdns.h”

struct in6_addr     src       = {{{0x20,0x1,0xdb, 0x8,0x0,0x0,0x0,
                                   0x0,0x0,0x8,0x8,0x0,0x20,
                                   0xc,0x41,0x7a}}};;
char                  dst[INET6_ADDRSTRLEN];
int32                 af       = AF_INET6;
uint32                dst_size = INET6_ADDRSTRLEN;
int16                 dss_errno;
int32                 ret;

/*-------------------------------------------------------------------
  To convert the IPv6 address from the sixteen byte numerical form
  given in Addr argument to the presentation form
  “2001:DB8::8:800:200C:417A
-------------------------------------------------------------------*/
ret = dss_inet_ntop( &src, af, dst, dst_size, &dss_errno );

if(ret == DSS_SUCCESS)
{
  MSG_MED( “dss_inet_ntop() returned success”, 0, 0, 0 );
}
else
{
  MSG_HIGH( “dss_inet_ntop() returned error %d”, dss_errno, 0, 0 );
}
```

3.11.5 Byte-ordering macros

The sockets API provides the macros to the calling applications listed in Table 3-3. These macros handle the potential byte order differences among different computer architectures and network protocols. All operate on unsigned and signed integer values.

Table 3-3 Sockets API interface macros

Macro	Description
`dss_htonl()`	Converts host-to-network long integer
`dss_htons()`	Converts host-to-network short integer
`dss_ntohl()`	Converts network-to-host long integer
`dss_ntohs()`	Converts network-to-host short integer

3.11.5.1 dss_htonl()

This function converts a host-to-network long integer.

Parameters

```
uint32 dss_htonl (
                    uint32      hostlong
                    )
```

→	`hostlong`	Unsigned 32-bit quantity to be converted from host-byte ordering to network-byte ordering

3.11.5.2 dss_htons()

This function converts a host-to-network short integer.

Parameters

```
uint32 dss_htons (
                    uint16      hostshort
                    )
```

→	`hostshort`	Unsigned 16-bit quantity to be converted from host-byte ordering to network-byte ordering

3.11.5.3 dss_ntohl()

This function converts a network-to-host long integer.

Parameters

```
uint32 dss_ntohl (
                    uint32      netlong
                    )
```

→	netlong	Unsigned 32-bit quantity to be converted from network-byte ordering to host-byte ordering

3.11.5.4 dss_ntohs()

This function converts a network-to-host short integer.

Parameters

```
uint32 dss_ntohs (
                    uint16      netshort
                    )
```

→	netshort	Unsigned 16-bit quantity to be converted from network-byte ordering to host-byte ordering

4 Network Subsystem and Interface Control API

This chapter provides specifications for the network subsystem and interface control API functions. Section 4.1 describes the functions related to the network subsystem, and Section 4.2 describes the interface control functions.

4.1 Network subsystem

Table 4-1 lists the calls related to the network subsystem.

Table 4-1 Network subsystem calls

Function	Description
`dss_get_app_net_policy()`	Returns the current network policy of the specified application
`dss_get_iface_status()`	Retrieves interface-specific information and stores it in the given interface control block
`dss_init_net_policy_info()`	Initializes the network policy information structure
`dss_netstatus()`	Reports the status of the network subsystem
`dsnet_stop()`	Closes the network subsystem
`dsnet_start()`	Brings up the network subsystem
`dss_set_app_net_policy()`	Sets the current network policy of the specified application
`dss_last_netdownreason()`	Returns the down reason for the last network interface to which the application was bound
`dss_get_app_profile_id()`	Returns the profile associated with an application

4.1.1 dss_init_net_policy_info()

This function initializes the network policy information structure passed to the dsnet_get_handle() function.

Parameters

```
void dss_init_net_policy_info(
                        dss_net_policy_info_type     *policy_info_ptr
                        )
```

→	dss_init_net_policy_info		
	←	policy_info_ptr	Pointer to the policy information structure to be initialized

Dependencies

This function must be invoked before specifying network policy to dsnet_get_handle() or dss_set_app_net_policy.

Description

This function initializes the network policy data structure pointed to by policy_info_ptr. An application must call this function before supplying network policy to dsnet_get_handle(). However, the application can modify different fields of the data structure after it is initialized.

Example

```
dss_net_policy_info my_net_policy;
sint15 errno;

/*------------------------------------------------------------------
Initialize the network policy and then modify it such that the
subsequent dsnet_start() selects a default interface only if it is already
up.
------------------------------------------------------------------*/
dss_init_net_policy_info(&my_net_policy);
my_net_policy.policy_flag = DSS_IFACE_POLICY_UP_ONLY;
dsnet_get_handle(my_net_cb, NULL, my_sock_cb, NULL, &my_policy_info,
&errno)
```

4.1.2 dsnet_get_policy()

This function returns the current network policy of the specified application.

Parameters

```
sint15 dsnet_get_policy (
                          sint15                    nethandle,
                          dss_net_policy_info_type  *policy_info_ptr,
                          sint15                    *errno
                          )
```

→	dsnet_get_policy		
	→	nethandle	Network handle
	←	policy_info_ptr	Pointer to the policy information structure
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADAPP – Invalid network handle specified ▪ DS_EFAULT – policy_info_ptr is NULL

Return value

This function returns:

- On error, DSS_ERROR is returned and the corresponding error value is placed in *errno.
- On success, DSS_SUCCESS is returned and the network policy of the specified application is returned in *policy_info_ptr.

Description

This function fills in the network policy data structure pointed to by policy_info_ptr with the current network policy of the application.

Example

```
sint15 nethandle;
dss_net_policy_info my_net_policy;
sint15 errno;

/*-----------------------------------------------------------------
Get the network policy of the application.
-----------------------------------------------------------------*/
if (dsnet_get_policy(nethandle, &my_net_policy, &errno) == DSS_ERROR)
{
MSG_ERROR("Error %d occurred, getting net policy of app=%d", errno,
nethandle, 0);
return;
}
```

4.1.3 dsnet_set_policy()

This function sets the network policy of the specified application. The network policy is used to select the network interface which will be brought up when dsnet_start() is called.

Parameters

```
sint15 dsnet_set_policy (
                          sint15                    nethandle,
                          dss_net_policy_info_type  *policy_info_ptr,
                          sint15                    *errno
                          )
```

→	dsnet_get_policy		
	→	nethandle	Network handle
	→	policy_info_ptr	Pointer to the policy information structure
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADAPP – Invalid network handle specified ▪ DS_EFAULT – Policy_info_ptr is pointing to invalid policy information

Return value

This function returns:

- On error, DSS_ERROR is returned and the error condition value is placed in *errno.
- On success, DSS_SUCCESS is returned.

Dependencies

This function should be called before calling dsnet_start() for the specified network policy to take effect.

The caller must initialize the *policy_info_ptr being passed to this function by calling dss_init_net_policy_info().

Description

This function sets the application network policy to the value passed in *policy_info_ptr. It is mandatory for an application to initialize *policy_info_ptr, passed to the dsnet_set_policy() function, by calling dss_init_net_policy_info(). The application can modify various fields of *policy_info_ptr after such an initialization.

An application not interested in a specific network policy may also supply NULL as policy_info_ptr; in this case the default network policy is used.

Example

```
sint15 nethandle;
dss_net_policy_info my_net_policy;
sint15 errno;

nethandle = dss_open_netlib(my_net_cb, my_sock_cb, &errno);
if (nethandle == DSS_ERROR)
{
MSG_ERROR("Could not open network library, error=%d", errno,0,0);
}
else
{
MSG_MED("Successfully opened network library, nethandle=%d",
nethandle,0,0);
}

/*-------------------------------------------------------------------
Initialize the network policy and then modify it such that the
subsequent dsnet_start() selects a default interface only if it is already
up.
-------------------------------------------------------------------*/
dss_init_net_policy_info(&my_net_policy);
my_net_policy.policy_flag = DSS_IFACE_POLICY_UP_ONLY;
if (dsnet_set_policy(nethandle, &my_net_policy, &errno) == DSS_ERROR)
{
  MSG_ERROR(“Failed to set the net policy of app=%d, error=%d”,
  nethandle, errno, 0);
}
```

4.1.4 dsnet_start()

This function starts up the network subsystem (CDMA Data Services and PPP) over the U_m interface for all sockets.

Parameters

```
sint15 dsnet_start (
                      sint15        nethandle,
                      sint15        *errno
                      )
```

→	dsnet_start		
	→	nethandle	Network handle
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EWOULDBLOCK – Operation would block ▪ DS_EBADAPP – Invalid network handle specified ▪ DS_ENETCLOSEINPROGRESS – Network close is in progress; application should call dsnet_start()only after the close has completed ▪ DS_EOPNOTSUPP – Mobile is in laptop call; mobile does not currently support simultaneous laptop and sockets calls; application should try to open PPP after some time; application will not get notification when laptop call ends

Return value

This function returns:

- DSS_SUCCESS – Successful operation (indicating the network subsystem has already been established)
- DSS_ERROR – Error; error condition value is placed in *errno

Dependencies

dsnet_start() cannot be called by the application if the network is in the process of closing. The network layer cannot queue the open request until the close is completely finished. Therefore, the application should wait for the network callback function to be called (after dsnet_stop() has completed) before making a call to dsnet_start(). A valid application must be specified as a parameter, obtained by a successful return of dss_open_netlib().

Description

An initial call to dsnet_start() returns DS_EWOULDBLOCK.

When the socket is connected or the network library stops trying to connect (PPP connection refused, service option denied, unable to set up traffic channel, etc.), the application callback function is invoked to provide event notification. The application may then query the network state through a call to dss_netstatus().

Calls to dsnet_start() while connection establishment is in progress or is already connected results in the Application Callback function being invoked. Again, the application queries the network state through dss_netstatus().

An application's netpolicy (dss_net_policy_info_type) has two fields, a unique identifier and an operational profile. These are used to retrieve the application's priority at connection establishment time. This priority value is used to control the usage of an interface by multiple applications. Arbitration is performed among applications contending for a particular interface where a higher priority application is allowed to preempt lower priority application(s). The arbiter checks all interfaces where the incoming application is requesting to establish a connection. Then the arbiter preempts the interface having the least priority application(s). As a result, lower priority applications will experience an abrupt loss of connection.

Example

Following is an example of this network callback for REX, specified and bound to the application in dss_open_netlib():

```
void app_dss_net_cb
(
   void* arg1                /*dummy arg; we don't use it*/
)
{
   (void)rex_set_sigs(&app_tcb,APP_NETWORK_CB_SIG);
}
```

Where:

- APP_NETWORK_CB_SIG = REX signal that is used for the asynchronous notification of network events
- app_tcb = specifies the application's task control block

Example

```
ret_val = dsnet_start(nethandle,&error_num);
```

Where:

- nethandle = network handle obtained by opening the network library with dss_open_netlib()

4.1.5 dsnet_stop()

This function initiates termination to bring down PPP and the traffic channel.

Parameters

```
sint15 dsnet_stop (
                      sint15    nethandle,
                      sint15    *errno
                      )
```

→	dss_netclose		
	→	nethandle	Network handle
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EWOULDBLOCK – Operation would block ▪ DS_EBADAPP – Invalid network handle specified ▪ DS_ENETCLOSEINPROGRESS – Network subsystem termination is in progress; a call to dsnet_stop() has already been issued

Return value

This function returns:

- DSS_SUCCESS – Successful operation (indicating the network subsystem has already been closed)
- DSS_ERROR – Error; error condition value is placed in *errno

Description

This function initiates termination to bring down PPP and the traffic channel. Upon successful close of the network subsystem, this function invokes the network callback function.

Example

```
void app_dss_net_cb
(
   void* arg1               /*dummy arg; we do not use it*/
)
{
   (void)rex_set_sigs(&app_tcb,APP_NETWORK_CB_SIG);
}
```

Where:

- APP_NETWORK_CB_SIG = REX signal that is used for the asynchronous notification for network-related events
- app_tcb = specifies the application's task control block

Example

Following is an example usage of this function call:

```
ret_val = dsnet_stop(nethandle,&error_num);
```

Where:

nethandle = network handle obtained by opening the network library with dss_open_netlib()

4.1.6 dss_get_iface_status()

The use of this function has been deprecated. The semantics of this function are identical to those of dsiface_get_status().

4.1.7 dss_netstatus()

This function returns the network subsystem status.

Parameters

```
sint15 dss_netstatus (
                        sint15       nethandle,
                        sint15       *errno
                        )
```

→	dss_netstatus		
	→	nethandle	Network handle
	←	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADAPP – Invalid network handle is specified ▪ DS_ENETNONET – Underlying network subsystem is unavailable ▪ DS_ENETISCONN – Network subsystem is established and available ▪ DS_ENETINPROGRESS – Network subsystem establishment is currently in progress ▪ DS_ENETCLOSEINPROGRESS – Network subsystem termination is currently in progress

Return value

This function returns:

- DSS_ERROR and places the error condition value in *errno

Description

This function provides the status of the network subsystem. It is called in response to DS_ENETDOWN errors.

NOTE: The origination status is based on the last attempted origination.

Example

```
ret_val = dss_netstatus(nethandle, &error_num);
```

4.1.8 dss_last_netdownreason()

This function returns the down reason for the last network interface to which the application was bound.

Parameters

```
sint15  dss_last_netdownreason (
                                  sint15                    nethandle,
                                  dss_net_down_reason_type  *reason,
                                  sint15                    *dss_errno
                                 )
```

→	dss_last_netdownreason		
	→	nethandle	Network handleentifier
	←	reason	Pointer to the returned last network down reason that is passed by reference; the following values are currently supported; more values may be added in the future: typedef enum { DSS_NET_DOWN_REASON_NOT_SPECIFIED, DSS_NET_DOWN_REASON_CLOSE_IN_PROGRESS, DSS_NET_DOWN_REASON_NW_INITIATED_TERMINATION, DSS_NET_DOWN_REASON_INSUFFICIENT_RESOURCES, DSS_NET_DOWN_REASON_UNKNOWN_APN, DSS_NET_DOWN_REASON_UNKNOWN_PDP, DSS_NET_DOWN_REASON_AUTH_FAILED, DSS_NET_DOWN_REASON_GGSN_REJECT, DSS_NET_DOWN_REASON_ACTIVATION_REJECT, DSS_NET_DOWN_REASON_OPTION_NOT_SUPPORTED, DSS_NET_DOWN_REASON_OPTION_UNSUBSCRIBED, DSS_NET_DOWN_REASON_OPTION_TEMP_OOO, DSS_NET_DOWN_REASON_NSAPI_ALREADY_USED, DSS_NET_DOWN_REASON_REGULAR_DEACTIVATION, DSS_NET_DOWN_REASON_QOS_NOT_ACCEPTED, DSS_NET_DOWN_REASON_NETWORK_FAILURE, DSS_NET_DOWN_REASON_UMTS_REATTACH_REQ, DSS_NET_DOWN_REASON_TFT_SEMANTIC_ERROR, DSS_NET_DOWN_REASON_TFT_SYNTAX_ERROR, DSS_NET_DOWN_REASON_UNKNOWN_PDP_CONTEXT, DSS_NET_DOWN_REASON_FILTER_SEMANTIC_ERROR, DSS_NET_DOWN_REASON_FILTER_SYNTAX_ERROR, DSS_NET_DOWN_REASON_PDP_WITHOUT_ACTIVE_TFT, DSS_NET_DOWN_REASON_INVALID_TRANSACTION_ID, DSS_NET_DOWN_REASON_MESSAGE_INCORRECT_SEMANTIC, DSS_NET_DOWN_REASON_INVALID_MANDATORY_INFO, DSS_NET_DOWN_REASON_MESSAGE_TYPE_UNSUPPORTED, DSS_NET_DOWN_REASON_MSG_TYPE_NONCOMPATIBLE_STATE, DSS_NET_DOWN_REASON_UNKNOWN_INFO_ELEMENT, DSS_NET_DOWN_REASON_CONDITIONAL_IE_ERROR,

			DSS_NET_DOWN_REASON_MSG_AND_PROTOCOL_STATE_UNCOMPATIBLE, DSS_NET_DOWN_REASON_PROTOCOL_ERROR, DSS_NET_DOWN_REASON_UNKNOWN_CAUSE_CODE, DSS_NET_DOWN_REASON_INTERNAL_MIN, DSS_NET_DOWN_REASON_INTERNAL_ERROR, DSS_NET_DOWN_REASON_INTERNAL_CALL_ENDED, DSS_NET_DOWN_REASON_INTERNAL_UNKNOWN_CAUSE_CODE, DSS_NET_DOWN_REASON_INTERNAL_MAX, DSS_NET_DOWN_REASON_MAX, EAP_CLIENT_ERR_UNABLE_TO_PROCESS, EAP_CLIENT_ERR_UNSUPPORTED_VERS, EAP_CLIENT_ERR_INSUFFICIENT_CHALLENGES, EAP_CLIENT_ERR_RAND_NOT_FRESH, EAP_NOTIFICATION_GENERAL_FAILURE_AFTER_AUTH, EAP_NOTIFICATION_GENERAL_FAILURE_BEFORE_AUTH, EAP_NOTIFICATION_TEMP_DENIED_ACCESS, EAP_NOTIFICATION_USER_NOT_SUBSCRIBED, EAP_SUCCESS, EAP_NOTIFICATION_REALM_UNAVAILABLE, EAP_NOTIFICATION_USER_NAME_UNAVAILABLE, EAP_NOTIFICATION_CALL_BARRED }dss_net_down_reason_type; A brief description of each of these enum values follows. The UMTS specific values come from [**Error! Reference source not found.**].
	←	dss_errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EBADAPP – Invalid network handle is specified ▪ DS_EFAULT – Invalid memory address

- DSS_NET_DOWN_REASON_CLOSE_IN_PROGRESS – Network session initiation request rejected as the previous session on the same network interface is currently being torn down.
- DSS_NET_DOWN_REASON_NW_INITIATED_TERMINATION – This cause code indicates that the session was terminated by the network.
- DSS_NET_DOWN_REASON_INSUFFICIENT_RESOURCES –PDP context activation request, secondary PDP context activation request or PDP context modification request cannot be accepted due to insufficient resources.
- DSS_NET_DOWN_REASON_UNKNOWN_APN – Requested service was rejected by the external packet data network because the access point name was not included although required or if the access point name could not be resolved.
- DSS_NET_DOWN_REASON_UNKNOWN_PDP – Requested service was rejected by the external packet data network because the PDP address or type could not be recognized.
- DSS_NET_DOWN_REASON_AUTH_FAILED –Requested service was rejected by the external packet data network due to a failed user authentication.
- DSS_NET_DOWN_REASON_GGSN_REJECT – GGSN has rejected the activation request.

- DSS_NET_DOWN_REASON_ACTIVATION_REJECT – Activation request rejected with unspecified reason.
- DSS_NET_DOWN_REASON_OPTION_NOT_SUPPORTED – The requested service option is not supported by the PLMN.
- DSS_NET_DOWN_REASON_OPTION_UNSUBSCRIBED – The requested service option is not subscribed for.
- DSS_NET_DOWN_REASON_OPTION_TEMP_OOO – MSC cannot service the request because of temporary outage of one or more functions required for supporting the service.
- DSS_NET_DOWN_REASON_NSAPI_ALREADY_USED – NSAPI requested by the MS in the PDP context activation request is already used by another active PDP context of this MS
- DSS_NET_DOWN_REASON_REGULAR_DEACTIVATION – Regular MS or network initiated PDP context deactivation.
- DSS_NET_DOWN_REASON_QOS_NOT_ACCEPTED – The new QoS cannot be accepted by the UE that were indicated by the network in the PDP Context Modification procedure.
- DSS_NET_DOWN_REASON_NETWORK_FAILURE –PDP context deactivation is caused by an error situation in the network.
- DSS_NET_DOWN_REASON_UMTS_REATTACH_REQ – This cause code is used by the network to request a PDP context reactivation after a GGSN restart. It is up to the application to reattach. The specification does not mandate it.
- DSS_NET_DOWN_REASON_TFT_SEMANTIC_ERROR – There is a semantic error in the TFT operation included in a secondary PDP context activation request or an MS-initiated PDP context modification.
- DSS_NET_DOWN_REASON_TFT_SYNTAX_ERROR – There is a syntactical error in the TFT operation included in a secondary PDP context activation request or an MS-initiated PDP context modification.
- DSS_NET_DOWN_REASON_UNKNOWN_PDP_CONTEXT – PDP context identified by the Linked TI IE the secondary PDP context activation request is not active.
- DSS_NET_DOWN_REASON_FILTER_SEMANTIC_ERROR – There is one or more semantic errors in packet filter(s) of the TFT included in a secondary PDP context activation request or an MS-initiated PDP context modification.
- DSS_NET_DOWN_REASON_FILTER_SYNTAX_ERROR – There is one or more syntactical errors in packet filter(s) of the TFT included in a secondary PDP context activation request or an MS-initiated PDP context modification.
- DSS_NET_DOWN_REASON_PDP_WITHOUT_ACTIVE_TFT – The network has already activated a PDP context without TFT.
- DSS_NET_DOWN_REASON_INVALID_TRANSACTION_ID – The equipment sending this cause has received a message with a transaction identifier which is not currently in use on the MS-network interface.
- DSS_NET_DOWN_REASON_MESSAGE_INCORRECT_SEMANTIC – Message is semantically incorrect.
- DSS_NET_DOWN_REASON_INVALID_MANDATORY_INFO – Mandatory information is invalid.

- DSS_NET_DOWN_REASON_MESSAGE_TYPE_UNSUPPORTED – Message type is non-existent or is not supported.
- DSS_NET_DOWN_REASON_MSG_TYPE_NONCOMPATIBLE_STATE – Message type is not compatible with the protocol state.
- DSS_NET_DOWN_REASON_UNKNOWN_INFO_ELEMENT – Information element is non-existent or is not implemented.
- DSS_NET_DOWN_REASON_CONDITIONAL_IE_ERROR – Conditional IE Error
- DSS_NET_DOWN_REASON_MSG_AND_PROTOCOL_STATE_UNCOMPATIBLE – Message is not compatible with the current protocol state.
- DSS_NET_DOWN_REASON_PROTOCOL_ERROR – Used to report a protocol error event only when no other cause in the protocol error class applies.
- DSS_NET_DOWN_REASON_UNKNOWN_CAUSE_CODE – Call is terminated but the cause is not mapped to a network down reason yet.
- DSS_NET_DOWN_REASON_INTERNAL_MIN – Used for range checking. Indicates the minimum value for the network down reasons which are generated because the call is terminated by the mobile.
- DSS_NET_DOWN_REASON_INTERNAL_ERROR – Call is terminated because of some internal error.
- DSS_NET_DOWN_REASON_INTERNAL_CALL_ENDED – Call is terminated locally by the mobile.
- DSS_NET_DOWN_REASON_INTERNAL_UNKNOWN_CAUSE_CODE – Call is terminated locally by the mobile but the cause is not mapped to a network down reason yet.
- DSS_NET_DOWN_REASON _EAP_CLIENT_ERR_UNABLE_TO_PROCESS – A general error code indicating failure.
- DSS_NET_DOWN_REASON _EAP_CLIENT_ERR_UNSUPPORTED_VERS – The peer does not support any of the versions listed in AT_VERSION_LIST.
- DSS_NET_DOWN_REASON _EAP_CLIENT_ERR_INSUFFICIENT_CHALLENGES – The peer's policy requires more triplets than the server included in AT_RAND.
- DSS_NET_DOWN_REASON _EAP_CLIENT_ERR_RAND_NOT_FRESH – The peer believes that the RAND challenges included in AT_RAND were not fresh.
- DSS_NET_DOWN_REASON_EAP_NOTIFICATION_GENERAL_FAILURE_AFTER_AUTH – Notification code indicating general failure after authentication.
- DSS_NET_DOWN_REASON_EAP_NOTIFICATION_GENERAL_FAILURE_BEFORE_AUTH – Notification code indicating general failure before authentication.
- DSS_NET_DOWN_REASON _EAP_NOTIFICATION_TEMP_DENIED_ACCESS – User has been temporarily denied access to the requested service. This implies failure and is used after successful authentication.
- DSS_NET_DOWN_REASON _EAP_NOTIFICATION_USER_NOT_SUBSCRIBED – User has not subscribed to the requested service. This implies failure and is used after successful authentication.
- DSS_NET_DOWN_REASON _EAP_SUCCESS –User has been successfully authenticated.

- DSS_NET_DOWN_REASON _EAP_NOTIFICATION_REALM_UNAVAILABLE – Notification error code while parsing an EAP identity indicating that NAI realm portion is unavailable in environments where a realm is required.
- DSS_NET_DOWN_REASON _EAP_NOTIFICATION_USER_NAME_UNAVAILABLE – Notification error code while parsing an EAP identity indicating that username is unavailable.
- DSS_NET_DOWN_REASON _EAP_NOTIFICATION_CALL_BARRED – Notification error code indicating that the call has been barred.
- DSS_NET_DOWN_REASON_INTERNAL_MAX – Used for range checking. Indicates the maximum value for the network down reasons that are generated because the call is terminated by the mobile.

Return value

This function returns:

- DSS_SUCCESS – Successful operation
- DSS_ERROR – Error; error condition value is placed in *dss_errno

Description

The application can call this function to retrieve the reason due to which the last network interface, to which the app was bound to, went down. On successful operation the value is passed back to the application in the reason parameter.

Example

The function can be called by the application as follows:

```
dss_net_down_reason_type    reason;
sint15                                          error_num;

retval = dss_last_netdownreason( nethandle, &reason, &error_num);
if (retval = = DSS_ERROR)
{
    MSG_HIGH("Could not return the last net down reason");
    return (DSS_ERROR);
}
else
{
   MSG_HIGH("The reason for last network down is: %d", reason);
   return (DSS_SUCCESS);
}

 Where:

nethandle = network handle obtained by opening the network library with
dss_open_netlib()
```

4.1.9 dss_get_app_profile_id()

This function returns the profile value associated with an application.

Parameters

```
int32 dss_get_app_profile_id (
                                uint32        app_type
                                )
```

→	dss_get_app_profile_id		
	→	app_type	Unsigned value of application identifier

Return value

This function returns:

- On success, profile of the application.
- On failure, PS_POLICY_MGR_PROFILE_INVALID (-1).

Description

This function invokes ps_policy_mgr_get_profile() to retrieve the profile value of an application based on its identifier.

Example

```
dss_net_policy_info my_net_policy;
uint32 app_identifier = MY_APP_IDENTIFIER;
sint15 errno;

/*-------------------------------------------------------------------
Initialize the network policy and then modify it such that the subsequent
dsnet_start() selects a default interface only if it is already up.
-------------------------------------------------------------------*/
dss_init_net_policy_info(&my_net_policy);
my_net_policy.policy_flag = DSS_IFACE_POLICY_UP_ONLY;
my_net_policy.cdms.data_session_profile_id =
dss_get_app_profile_id(app_identifier);

if (my_net_policy.cdms.data_session_profile_id >= 0)
{
  dsnet_get_handle(my_net_cb, NULL, my_sock_cb, NULL, &my_policy_info,
&errno);
}
```

4.2 Interface control

Table 4-2 lists the calls related to the interface control.

Table 4-2 Interface control calls

Function	Description
`dss_get_iface_id()`	Provides the interface ID being used by an application
`dss_get_iface_id_by_policy()`	Provides the interface ID based on specified network policy
`dss_iface_ioctl()`	Performs a variety of control functions on specified interface

4.2.1 dss_get_iface_id()

This function is used to get the identifier of the interface being used by an application or socket.

Parameters

```
dss_iface_id_type dss_get_iface_id (
                                sint15          nethandle
                                )
```

→	`dss_get_iface_id`		
	→	`nethandle`	Network handle

Return value

This function returns:

- DSS interface identifier corresponding to the supplied information – In case of success
- DSS_IFACE_INVALID_ID – In case of failure

Description

This function is used by applications to get a handle to the interface they are using. The interface can be specified using a nethandle. If dsnet_start() has already been called by the application, then the interface returned is the one that the application is currently using.

If the application has not called dsnet_start(), then this function does a routing lookup and returns the interface that the routing layer selects based on the policy specified in the application control block. Subsequent calls to dsnet_start() will result in a fresh routing lookup and will bring up the most-preferred interface based on the policy specified in the application control block. The interface brought up by dsnet_start might differ from the interface returned by the previous dss_get_iface_id call. If the application wants to bring up the same interface that is returned by the dss_get_iface_id_call(), it should call dss_set_app_netpolicy() with the iface_id returned before making any subsequent calls to dss_get_iface_id() or dsnet_start().

Example

```
sint15 nethandle = my_nethandle;
dss_iface_id_type iface_id;
if ((iface_id = dss_get_iface_id(nethandle)) == DSS_IFACE_INVALID_ID)
{
  MSG_ERROR(“Failed to get the iface id for nethandle=%d”, my_nethandle, 0,
0);
}
```

4.2.2 dss_get_iface_id_by_policy()

This function is used to get the interface identifier based on network policy.

Parameters

```
dss_iface_id_type dss_get_iface_id_by_policy (
                            dss_net_policy_info_type     net_policy
                            sint15 *  errno
                            )
```

→	dss_get_iface_id_by_policy		
	→	net_policy	Specifies the network policy to be used for determining the interface identifier
	→	errno	Pointer to the returned error value that is passed by reference; valid values are: ▪ DS_EFAULT – Network policy structure is not initialized ▪ DS_ENOROUTE – No interface can be determined from the network policy

Return value

This function returns:

- DSS interface identifier corresponding to the supplied information – In case of success
- DSS_IFACE_INVALID_ID – In case of failure

Description

This function is used by applications to get a handle to the interface. The interface is determined based on the specified network policy.

Example

```
dss_net_policy_info_type netpolicy;
dss_iface_id_type iface_id;
sint15 errno;

dss_init_netpolicy_info(&netpolicy);
netpolicy.policy_flag = DSS_IFACE_POLICY_UP_ONLY
netpolicy.iface.info.name = DSS_IFACE_CDMA_SN
if ((iface_id = dss_get_iface_id_by_policy(netpolicy, &errno)) ==
DSS_IFACE_INVALID_ID)
{
  MSG_ERROR("Failed to get the iface id for errno=%d", errno, 0, 0);
}
```

4.2.3 dss_get_iface_id_by_qos_handle()

This function is used to get the interface identifier based on the QoS handle.

Parameters

```
dss_iface_id_type dss_get_iface_id_by_qos_handle (
   dss_iface_ioctl_qos_handle_type handle
   )
```

→	dss_get_iface_id_by_qos_handle		
	→	handle	Specifies the QoS handle that is returned as a parameter to the DSS_IFACE_IOCTL_QOS_REQUEST or in a Network initiated QoS CB notification following registration via DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST

Return value

This function returns:

- DSS interface identifier corresponding to the supplied information – In case of success
- DSS_IFACE_INVALID_ID – In case of failure

Description

This function is used by applications to get a handle to the interface and corresponding secondary flow or physical link based on the QoS handle previously obtained. The secondary flow or link is the one providing the desired QoS previously requested indicated by the specified QoS handle. The interface handle can be used by the application for subsequent operations on the flow or link, e.g., for registering a PHYS_LINK event.

Example

```
dss_iface_ioctl_qos_handle_type handle;
dss_iface_id_type iface_id;

/* The qos handle is obtained as part of the QoS request IOCTL operation
*/

if ((iface_id = dss_get_iface_id_by_qos_handle(handle)) ==
DSS_IFACE_INVALID_ID)
{
  MSG_ERROR("Failed to get the iface id for qos handle %d", handle, 0, 0);
}
```

4.2.4 dss_iface_ioctl()

This function performs a variety of control functions on interfaces.

NOTE: This feature is supported only in multimode architecture, i.e., only if FEATURE_DATA is defined.

Parameters

```
int dss_iface_ioctl
                    (
                    dss_iface_id_type iface_id,
                    dss_ioctl_type    ioctl_name,
                    void              *argval_ptr,
                    sint15            *dss_errno
                    )
```

→	dss_iface_ioctl		
	→	iface_id	Identifier of the interface on which the control function is to be performed Applications can obtain an interface identifier by calling dss_get_iface_id() or through the parameter supplied to the network event callback registered with the dsnet_get_handle() function call
	→	ioctl_name	Selects the control function to be performed on the specified interface
	↔	argval_ptr	Pointer to a structure that contains additional information for performing the specified operation in case of a write/set operation or contains information returned to the caller in case of a read/get operation
	←	dss_errno	Pointer to the returned error value that is passed by reference; the following are valid values for errno (defined in dserrno.h): ▪ DS_EBADF – Returned by dss_iface_ioctl() if the specified id_ptr is invalid, i.e., id_ptr does not map to a valid ps_iface_ptr ▪ DS_EINVAL – Returned by dss_iface_ioctl() when the specified IOCTL does not belong to the common set of IOCTLs and there is no IOCTL modehandler registered for the specified interface ▪ DS_EOPNOTSUPP – Returned by the lower level IOCTL mode handler when the specified IOCTL is not supported by the interface, i.e., certain iface-specific common IOCTLs (these are common IOCTLs, but the implementation is mode-specific, i.e., GO_DORMANT) ▪ DS_EFAULT – Specified arguments for the IOCTL are incorrect or if dss_iface_ioctl() or a mode handler encounters an error while executing the IOCTL, i.e., if the 1X interface cannot GO_DORMANT it would return this error ▪ DS_NOMEMORY – Insufficient buffers during execution

Return value

This function returns:

- 0 – In case of success
- 1 – In case of failure (dss_errno contains the error value)

Description

This function provides the ability to perform various control operations on a particular interface. The interface is specified using an interface name and instance combination. The argval_ptr points to a data type that contains data to perform the specified operation (each IOCTL has an associated type defined). In case of a set/write operation, this data type contains data that is passed in by the caller to perform the specified operation, and in case of a get/read operation, this data type contains data that is returned by this function after performing the specified operation. The dss_errno value returned has the error value in case of a failure.

This function performs IOCTL operations common to all interfaces and passes on interface-specific operations to the corresponding controllers. Table 4-3 lists the supported IOCTLs.

Table 4-3 List of supported IOCTLs

Name	Description
DSS_IFACE_IOCTL_GET_IPV4_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_IPV6_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_IPV4_PRIM_DNS_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_IPV6_PRIM_DNS_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_IPV4_SECO_DNS_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_IPV6_SECO_DNS_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_ALL_DNS_ADDRS	See Section 0
DSS_IFACE_IOCTL_GET_MTU	See Section 0
DSS_IFACE_IOCTL_GET_IP_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_STATE	See Section 0
DSS_IFACE_IOCTL_GET_PHYS_LINK_STATE	See Section 0
DSS_IFACE_IOCTL_REG_EVENT_CB	See Section 0
DSS_IFACE_IOCTL_DEREG_EVENT_CB	See Section 0
DSS_IFACE_IOCTL_GET_ALL_IFACES	See Section 0
DSS_IFACE_IOCTL_GET_HW_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_IFACE_NAME	See Section 0
DSS_IFACE_IOCTL_GET_IP_FAMILY	See Section 0
DSS_IFACE_IOCTL_GET_BEARER_TECHNOLOGY	See Section 0
DSS_IFACE_IOCTL_GET_DATA_BEARER_RATE	See Section 0
DSS_IFACE_IOCTL_IS_LAPTOP_CALL_ACTIVE	See Section 0
DSS_IFACE_IOCTL_GET_SIP_SERV_ADDR	See Section 0

Name	Description
DSS_IFACE_IOCTL_GET_SIP_SERV_DOMAIN_NAMES	See Section 0
DSS_IFACE_IOCTL_GENERATE_PRIV_IPV6_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_DOMAIN_NAME_SEARCH_LIST	See Section 0
DSS_IFACE_IOCTL_REFRESH_DHCP_CONFIG_INFO	See Section 0
DSS_IFACE_IOCTL_GET_DORMANCY_INFO_CODE	See Section 0
DSS_IFACE_IOCTL_QOS_GET_MODE	See Section 0
DSS_IFACE_IOCTL_QOS_REQUEST	See Section 0
DSS_IFACE_IOCTL_QOS_REQUEST_EX	See Section 0
DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST	See Section 0
DSS_IFACE_IOCTL_QOS_NET_INITIATED_RELEASE	See Section 0
DSS_IFACE_IOCTL_QOS_MODIFY	See Section 0
DSS_IFACE_IOCTL_QOS_RELEASE	See Section 0
DSS_IFACE_IOCTL_QOS_RELEASE_EX	See Section 0
DSS_IFACE_IOCTL_QOS_SUSPEND	See Section 0
DSS_IFACE_IOCTL_QOS_SUSPEND_EX	See Section 0
DSS_IFACE_IOCTL_QOS_RESUME	See Section 0
DSS_IFACE_IOCTL_QOS_RESUME_EX	See Section 0
DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY	See Section 0
DSS_IFACE_IOCTL_QOS_GET_STATUS	See Section 4.2.4.2.1
DSS_IFACE_IOCTL_QOS_GET_FLOW_SPEC	See Section 0
DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC	See Section 0
DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC2	See Section 0
DSS_IFACE_IOCTL_PRIMARY_QOS_GET_GRANTED_FLOW_SPEC	See Section 0
DSS_IFACE_IOCTL_QOS_GET_GRANTED_FILTER_SPEC	See Section 0
DSS_IFACE_IOCTL_GET_NETWORK_SUPPORTED_QOS_PROFILES	See Section 0
DSS_IFACE_IOCTL_ON_QOS_AWARE_SYSTEM	See Section 0
DSS_IFACE_IOCTL_GO_ACTIVE	See Section 0
DSS_IFACE_IOCTL_GO_DORMANT	See Section 0
DSS_IFACE_IOCTL_GO_NULL	See Section 0
DSS_IFACE_IOCTL_IPCP_DNS_OPT	See Section 0
DSS_IFACE_IOCTL_MT_REG_CB	See Section 0
DSS_IFACE_IOCTL_MT_DEREG_CB	See Section 0
DSS_IFACE_IOCTL_MCAST_JOIN	See Section 0
DSS_IFACE_IOCTL_MCAST_LEAVE	See Section 0
DSS_IFACE_IOCTL_MCAST_JOIN_EX	See Section 0

Name	Description
DSS_IFACE_IOCTL_MCAST_LEAVE_EX	See Section 0
DSS_IFACE_IOCTL_MCAST_REGISTER_EX	See Section 0
DSS_IFACE_IOCTL_707_GET_MDR	See Section 0
DSS_IFACE_IOCTL_707_SET_MDR	See Section 0
DSS_IFACE_IOCTL_707_GET_DORM_TIMER	See Section 0
DSS_IFACE_IOCTL_707_SET_DORM_TIMER	See Section 0
DSS_IFACE_IOCTL_707_GET_RLP_ALL_CURR_NAK	See Section 0
DSS_IFACE_IOCTL_707_SET_RLP_ALL_CURR_NAK	See Section 0
DSS_IFACE_IOCTL_707_GET_RLP_ALL_DEF_NAK	See Section 0
DSS_IFACE_IOCTL_707_SET_RLP_ALL_DEF_NAK	See Section 0
DSS_IFACE_IOCTL_707_GET_RLP_ALL_NEG_NAK	See Section 0
DSS_IFACE_IOCTL_707_GET_RLP_QOS_NA_PRI	See Section 0
DSS_IFACE_IOCTL_707_SET_RLP_QOS_NA_PRI	See Section 0
DSS_IFACE_IOCTL_707_SDB_SUPPORT_QUERY	See Section 0
DSS_IFACE_IOCTL_707_ENABLE_HOLDDOWN	See Section 0
DSS_IFACE_IOCTL_707_ENABLE_HDR_HPT_MODE	See Section 0
DSS_IFACE_IOCTL_707_ENABLE_HDR_REV0_RATE_INTERIA	See Section 0
DSS_IFACE_IOCTL_707_HDR_GET_RMAC3_INFO	See Section 0
DSS_IFACE_IOCTL_707_GET_TX_STATUS	See Section 0
DSS_IFACE_IOCTL_707_GET_SESSION_TIMER	See Section 0
DSS_IFACE_IOCTL_707_SET_SESSION_TIMER	See Section 0
DSS_IFACE_IOCTL_707_GET_INACTIVITY_TIMER	See Section 0
DSS_IFACE_IOCTL_707_GET_INACTIVITY_TIMER	See Section 0
DSS_IFACE_IOCTL_707_GET_HYSTERESIS_ACT_TIMER	See Section 0
DSS_IFACE_IOCTL_707_SET_HYSTERESIS_ACT_TIMER	See Section 0
DSS_IFACE_IOCTL_BCMCS_DB_UPDATE	See Section 0
DSS_IFACE_IOCTL_BCMCS_ENABLE_HANDOFF_REG	See Section 0
DSS_IFACE_IOCTL_BCMCS_BOM_CACHING_SETUP	See Section 0
DSS_IFACE_IOCTL_MBMS_MCAST_CONTEXT_ACTIVATE	See Section 0
DSS_IFACE_IOCTL_MBMS_MCAST_CONTEXT_DEACTIVATE	See Section 0
DSS_IFACE_IOCTL_ENABLE_FIREWALL	See Section 0
DSS_IFACE_IOCTL_DISABLE_FIREWALL	See Section 0
DSS_IFACE_IOCTL_ADD_FIREWALL_RULE	See Section 0
DSS_IFACE_IOCTL_DELETE_FIREWALL_RULE	See Section 0
DSS_IFACE_IOCTL_GET_FIREWALL_RULE	See Section 0
DSS_IFACE_IOCTL_GET_FIREWALL_TABLE	See Section 0

Name	Description
DSS_IFACE_IOCTL_ADD_STATIC_NAT_ENTRY	See Section 0
DSS_IFACE_IOCTL_DELETE_STATIC_NAT_ENTRY	See Section 0
DSS_IFACE_IOCTL_GET_STATIC_NAT_ENTRY	See Section 0
DSS_IFACE_IOCTL_GET_DYNAMIC_NAT_ENTRY_TIMEOUT	See Section 0
DSS_IFACE_IOCTL_SET_DYNAMIC_NAT_ENTRY_TIMEOUT	See Section 0
DSS_IFACE_IOCTL_SET_NAT_IPSEC_VPN_PASS_THROUGH	See Section 0
DSS_IFACE_IOCTL_GET_NAT_IPSEC_VPN_PASS_THROUGH	See Section 0
DSS_IFACE_IOCTL_DHCP_ARP_CACHE_UPDATE	See Section 0
DSS_IFACE_IOCTL_DHCP_ARP_CACHE_CLEAR	See Section 0
DSS_IFACE_IOCTL_DHCP_SERVER_GET_DEVICE_INFO	See Section 0
DSS_IFACE_IOCTL_ADD_DMZ	See Section 0
DSS_IFACE_IOCTL_GET_DMZ	See Section 0
DSS_IFACE_IOCTL_DELETE_DMZ	See Section 0
DSS_IFACE_IOCTL_GET_NAT_PUBLIC_IP_ADDR	See Section 0
DSS_IFACE_IOCTL_GET_IFACE_STATS	See Section 0
DSS_IFACE_IOCTL_RESET_IFACE_STATS	See Section 0
DSS_IFACE_IOCTL_GET_NAT_L2TP_VPN_PASS_THROUGH	See Section 0
DSS_IFACE_IOCTL_SET_NAT_L2TP_VPN_PASS_THROUGH	See Section 0
DSS_IFACE_IOCTL_GET_NAT_PPTP_VPN_PASS_THROUGH	See Section 0
DSS_IFACE_IOCTL_SET_NAT_PPTP_VPN_PASS_THROUGH	See Section 0

4.2.4.1 Common IOCTLs

These operations are supported by all interfaces.

DSS_IFACE_IOCTL_GET_IPV4_ADDR

This operation is used to get the IPV4 address of an interface. The data type used as argval for this operation is dss_iface_ioctl_ipv4_addr_type.

DSS_IFACE_IOCTL_GET_IPV6_ADDR

This operation is used to get the IPV6 address of an interface. The data type used as argval is dss_iface_ioctl_ipv6_addr_type.

DSS_IFACE_IOCTL_GET_IPV4_PRIM_DNS_ADDR

This operation is used to get the primary DNS (IPV4) address of an interface. The data type used as argval is dss_iface_ioctl_ipv4_prim_dns_addr_type.

DSS_IFACE_IOCTL_GET_IPV6_PRIM_DNS_ADDR

This operation is used to get the primary DNS (IPV6) address of an interface. The data type used as argval is dss_iface_ioctl_ipv6_prim_dns_addr_type.

DSS_IFACE_IOCTL_GET_IPV4_SECO_DNS_ADDR

This operation is used to get the secondary DNS (IPV4) address of an interface. The data type used as argval is dss_iface_ioctl_ipv4_seco_dns_addr_type.

DSS_IFACE_IOCTL_GET_IPV6_SECO_DNS_ADDR

This operation is used to get the secondary DNS (IPV6) address of an interface. The data type used as argval is dss_iface_ioctl_ipv6_seco_dns_addr_type.

DSS_IFACE_IOCTL_GET_ALL_DNS_ADDRS

This operation is used to get all the available DNS addresses on an interface. The data type used as argval is dss_iface_ioctl_get_all_dns_addrs_type. This structure includes a pointer, dns_addrs_ptr, that points to the application defined memory for the DNS addresses of type ip_addr_type and a num_dns_addrs used as an input/output parameter that specifies the number of addresses requested by the application as well as the actual number of addresses returned.

DSS_IFACE_IOCTL_GET_MTU

This operation is used to get the MTU of an interface. The data type used as argval is dss_iface_ioctl_mtu_type.

DSS_IFACE_IOCTL_GET_IP_ADDR

This operation is used to get the IP address of an interface. This IOCTL operates on either an IPv4 or IPv6 interface and returns the associated IP address. In the case of IPv6 interfaces, it returns the default (public) IP address. The data type used as argval is dss_iface_ioctl_get_ip_addr_type.

DSS_IFACE_IOCTL_GET_STATE

This operation is used to get the state of an interface. The data type used as argval is dss_iface_ioctl_state_type.

DSS_IFACE_IOCTL_GET_PHYS_LINK_STATE

This operation is used to get the physical link state of an interface. The data type used as argval is dss_iface_ioctl_phys_link_state_type.

DSS_IFACE_IOCTL_REG_EVENT_CB

This operation is used to register callbacks for events for an interface. The application can have only one callback per interface for each event. The data type used as argval is dss_iface_ioctl_ev_cb_type. This structure includes an application-specified callback of type dss_iface_ioctl_event_cb, a union of type dss_iface_ioctl_event_enum_type, a void * user_data_ptr, which is returned to the application as an argument in the callback, and a network handle. Applications can register for the following events:

- DSS_IFACE_IOCTL_DOWN_EV
- DSS_IFACE_IOCTL_UP_EV
- DSS_IFACE_IOCTL_COMING_UP_EV
- DSS_IFACE_IOCTL_GOING_DOWN_EV

- DSS_IFACE_IOCTL_PHYS_LINK_DOWN_EV
- DSS_IFACE_IOCTL_PHYS_LINK_UP_EV
- DSS_IFACE_IOCTL_PHYS_LINK_COMING_UP_EV
- DSS_IFACE_IOCTL_PHYS_LINK_GOING_DOWN_EV
- DSS_IFACE_IOCTL_ADDR_CHANGED_EV
- DSS_IFACE_IOCTL_BEARER_TECH_CHANGED_EV
- DSS_IFACE_IOCTL_707_NETWORK_SUPPORTED_QOS_PROFILES_CHANGED_EV
- DSS_IFACE_IOCTL_QOS_AWARE_SYSTEM_EV
- DSS_IFACE_IOCTL_QOS_UNAWARE_SYSTEM_EV
- DSS_IFACE_IOCTL_PREFIX_UPDATE_EV

DSS_IFACE_IOCTL_DEREG_EVENT_CB

This operation is used to deregister an event callback for an interface. The data type used as argval is dss_iface_ioctl_ev_cb_type. The application must specify a valid network handle when deregistering the callback.

DSS_IFACE_IOCTL_GET_ALL_IFACES

This operation is used to retrieve a list of the currently enabled interfaces. The data type used as argval is dss_iface_ioctl_all_ifaces_type. For this IOCTL, a NULL id_ptr has to be passed as a parameter.

DSS_IFACE_IOCTL_GET_HW_ADDR

This operation is used to get the hardware address associated with the interface. The data type used as argval is dss_iface_ioctl_hw_addr_type.

DSS_IFACE_IOCTL_GET_IFACE_NAME

This operation is used to retrieve the name of the associated interface. The data type used as argval is dss_iface_ioctl_iface_name_type.

DSS_IFACE_IOCTL_GET_IP_FAMILY

This operation is used to get the IP family of the associated interface. The data type used as argval is dss_iface_ioctl_ip_family_type.

DSS_IFACE_IOCTL_GET_BEARER_TECHNOLOGY

This operation is used to retrieve the underlying technology associated with the interface. The IOCTL also returns the negotiated service option. For example, this IOCTL returns if mobile is in an EV-DO Rev A network and also if MPA or EMPA is negotiated. The data type used as argval is dss_iface_ioctl_bearer_tech_type.

DSS_IFACE_IOCTL_GET_DATA_BEARER_RATE

This operation is used to retrieve the data bearer rate information of the associated interface. The data type used as argval is dss_iface_ioctl_data_bearer_rate.

DSS_IFACE_IOCTL_IS_LAPTOP_CALL_ACTIVE

This operation is used to check whether the associated interface is in a laptop call. The data type used as argval is dss_iface_ioctl_is_laptop_call_active_type.

DSS_IFACE_IOCTL_GET_SIP_SERV_ADDR

This operation is used to get the Session Initiation Protocol (SIP) server IP addresses for an interface. The data type used as argval for this operation is dss_iface_ioctl_sip_serv_addr_info_type.

DSS_IFACE_IOCTL_GET_SIP_SERV_DOMAIN_NAMES

This operation is used to get the SIP server domain names for an interface. The data type used as argval for this operation is dss_iface_ioctl_sip_serv_domain_name_info_type.

DSS_IFACE_IOCTL_GENERATE_PRIV_IPV6_ADDR

This operation is used to generate a private IPv6 address. The data type used as argval for this operation is dss_iface_ioctl_priv_ipv6_addr_type. This data type takes five different parameters. The first parameter is the network handle. The second is iid_params which contains a boolean is_unique. This boolean should be set to either FALSE if the application wants a private shareable address or TRUE if it wants a private unique address. The next parameters are an application-specified callback of type dss_iface_ioctl_event_cb and a void * user_data_ptr (which is returned to the application as an argument to the callback). The application must pass a valid callback as the IOCTL will automatically register the application for IPv6 privacy events for the address returned. The final parameter is an IP address which will be populated with the application's address if the operation was successful. If not, the address will be set to 0 and the IOCTL will return -1 and dss_ernno will be populated with the reason for failure. See Section 8.1 for more information on IPv6 privacy extension support.

DSS_IFACE_IOCTL_GET_DOMAIN_NAME_SEARCH_LIST

This operation is used to get the domain name search list for an interface (see [**Error! Reference source not found.**]) used by DHCPv6. The data type used as argval for this operation is dss_iface_ioctl_domain_name_search_list_type.

DSS_IFACE_IOCTL_REFRESH_DHCP_CONFIG_INFO

This operation is used to trigger a refresh of the DHCP configuration on an interface. It will expect a DSS_IFACE_IOCTL_EXTENDED_IP_CONFIG_EV event upon success or failure of this refresh. No data type is used as argval, and this can be passed as NULL.

DSS_IFACE_IOCTL_GET_DORMANCY_INFO_CODE

This IOCTL is used to get the reason code for dormancy. The data type used as argval for this operation is dss_iface_ioctl_dormancy_info_code_enum_type.

DSS_IFACE_IOCTL_QOS_GET_MODE

This IOCTL can be used to query the current QoS mode of the device which is determined by the network. The IOCTL returns one of the following status values: UE_ONLY, UE_AND_NW, NW_ONLY. The QoS mode can be one of these three types. UE_ONLY means only user equipment (mobile) can request QoS. UE_AND_NW means both the UE and network can request

QoS, whereas NW_ONLY applies only to the network. This IOCTL applies to all technologies, although it is currently implemented only by UMTS. The data type used as argval is dss_iface_ioctl_qos_mode_type as a parameter.

4.2.4.2 Interface-specific common IOCTLs

These IOCTLs could potentially be supported by more than one interface, and their implementation is interface-specific.

DSS_IFACE_IOCTL_QOS_REQUEST

This operation is used to request a specific QoS in either the receive, transmit, or both directions on an interface. QoS can only be requested on the interface with which the application has a binding. In other words, the application must have previously brought up the interface before requesting a specific QoS on that interface. The parameters to the IOCTL specify the flow specification and filter specification for each direction on which the QoS is requested. If the operation is successful, the IOCTL returns a handle to this QoS instance, which is used for all subsequent operations on this QoS instance. Each QoS instance is associated with a unique flow and the flow, in turn, is bound to one of the links of the iface on which the IOCTL is performed. The link provides the requested QoS characteristics to the traffic flow identified by the filter specification. This IOCTL needs to perform asynchronous operation and, hence, a return value of SUCCESS only means that the QoS request has been submitted. The final outcome is later indicated by a QoS event. For CDMA targets this operation will fail on a QoS-unaware system and will return DS_EQOSUNAWARE. The data type used as argval is dss_iface_ioctl_qos_request_type. For further details on the QoS architecture and usage model, see Chapter 6 of this document.

DSS_IFACE_IOCTL_QOS_REQUEST_EX

This IOCTL is an extension of DSS_IFACE_IOCTL_QOS_REQUEST. REQUEST_EX allows an application to bundle multiple QoS requests in the same signaling message. In addition, the application may specify an opcode, indicating whether it wishes to merely configure the QoS instances and grant them at a later time by calling QOS_RESUME (DSS_IFACE_IOCTL_QOS_CONFIGURE_OP) or store and grant the QoS immediately (DSS_IFACE_IOCTL_QOS_REQUEST_OP). QoS can be requested only on the interface (indicated by iface_id) to which the application (identified by nethandle) is bound (e.g., by calling dss_pppopen()). In other words, the application must have previously launched the interface before requesting any QoS on that interface. The QoS parameters to the IOCTL specify the flow and filter specifications for each direction in which the QoS is requested. If the operation is successful, the IOCTL returns a unique handle for each QoS instance that is used for all subsequent operations on the QoS instances. If QoS validation fails for any given QoS, all QoS instances are cleaned up, no QoS handles are returned, and the IOCTL returns an error. The application is required to allocate the necessary memory to store the returned QoS handles. This array of QoS handles must be equal to or greater than the number specified in num_qos_specs. Each QoS instance is associated with a unique flow, and the flow, in turn, is bound to one of the links of the iface on which the IOCTL is performed. The link provides the requested QoS characteristics to the traffic flow identified by the filter spec. This IOCTL needs to perform asynchronous operation and hence a return value of SUCCESS only means that the QoS requests have been submitted. The final outcome is later indicated by a QoS event for each requested QoS instance as described later. This IOCTL currently supports no more than 10 *bundled* requests and will return EINVAL if a number greater than that is set as num_qos_specs. The data type used as argval is dss_iface_ioctl_qos_request_ex_type. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST

This operation is used to register for notifications of Network Initiated QoS. This registration can only be requested on the interface with which the application has a binding. In other words, the application must have previously brought up the interface before requesting the registration. The parameters to the IOCTL include filter specification in either receive, transmit, or both directions of an interface. The application shall be notified only for Network Initiated QoS matching this filter specification (See Section 6.3.3.3 for more details of filter matching semantics). If the operation is successful, the IOCTL returns a handle representing the registration request. This handle may be used to cancel the registration.

If not matching QoS is available, this IOCTL returns DSS_EWOULDBLOCK. If a matching QoS already exists when this IOCTL is invoked, the corresponding QoS instance handle will be returned in the response to the application.

If a matching QoS is initiated by the network or if it already exists, the application CB shall be called. The CB parameters include the registration request handle as well as a handle to the matching QoS instance. The application can use the QoS instance handle for any QoS operations that provide information on the QoS instance, e.g., Section 4.2.4.2.1 DSS_IFACE_IOCTL_QOS_GET_STATUS, however, QoS operations that manipulate the QoS, such as Section 0 DSS_IFACE_IOCTL_QOS_MODIFY, are not applicable for Network Initiated QoS thus not supported. DSS_IFACE_IOCTL_QOS_RELEASE is supported for Network Initiated Qos. Application should call this API if it has finished using the Network Initiated Qos session and expects no further QoS events on the Network Initiated QoS handle. The application shall receive a new QoS handle in each notification of new Network Initiated QoS. The data type used as argval for this ioctl is dss_iface_ioctl_qos_net_initiated_request_type.

The application may specify an empty filters list as filters specification in either receive, transmit, or both directions. Such specification means that Network Initiated QoS with any filters specification in that direction matches the application request. If the application specifies an empty filters list for both receive and transmit directions it means that any Network Initiated QoS shall match the application request and the application CB shall be called when such QoS is initiated by the Network.

On DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST from application, the system shall immediately notify the application of any currently existing Network Initiated QoS instances that match the filters specification provided by the application. Such notifications shall be made via application CB provided in the request (i.e. in the same manner as notifications made for Network Initiated QoS that is created by the network after the application registration request has been made).

Once application called DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST, it remains registered for Network Initiated notifications as long as it does not cancel the registration or the network interface is UP (DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST is cancelled automatically when the network interface goes DOWN). To clarify, even if the application is notified of one Network Initiated QoS, it shall remain registered and shall get further notifications if the network initiates additional QoS instance(s).

For further details on the QoS architecture and usage model, refer to Chapter 6 of this document.

DSS_IFACE_IOCTL_QOS_NET_INITIATED_RELEASE

This operation is used to cancel application registration for Network Initiated QoS notifications applied through DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST.

Although this API cancels registration for new Network Initiated QoS notifications it does not effect registration to QoS events for existing Network Initiated QoS sessions the application has already been notified of. The application shall continue to receive QoS events for those QoS sessions until the network releases them or until the application releases them via DSS_IFACE_IOCTL_QOS_RELEASE.

The data type used as argval is dss_iface_ioctl_qos_net_initiated_release_type.

DSS_IFACE_IOCTL_QOS_MODIFY

This operation is used to modify the previously requested QoS in either the receive or transmit directions, or in both directions, or to request new QoS in a direction for which QoS was not requested previously. Either flow specification, filter specification, or both can be modified for a specified QoS instance in each direction. The QoS instance is identified by the handle previously returned by DSS_IFACE_IOCTL_QOS_REQUEST. This IOCTL needs to perform asynchronous operation and hence a return value of SUCCESS only means that the request to modify QoS has been submitted. The final outcome is later indicated by a QoS event. This IOCTL is supported only on UMTS targets. The data type used as argval is dss_iface_ioctl_qos_modify_type. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_RELEASE

This operation releases a previously requested QoS. The QoS instance is identified by the handle previously returned by DSS_IFACE_IOCTL_QOS_REQUEST or in a Network Initiated QoS CB notification following registration via DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST. For QoS instance handles obtained via DSS_IFACE_IOCTL_QOS_REQUEST this operation also releases the flow providing the QoS and makes it available for other QoS requests. In this case, the IOCTL needs to perform asynchronous operation; therefore, a return value of SUCCESS only means that the QoS request has been submitted. For Network Initiated QoS instance handles, this operation does not affect the flow providing the QoS. In this case the operation is synchronous and the application can expect to receive no further QoS events on the QoS instance handle. This IOCTL requires dss_iface_ioctl_qos_release_type as a parameter. For CDMA targets this operation can be performed even in a QoS-unaware system. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_RELEASE_EX

This IOCTL releases *n* previously requested QoS instances. The QoS instances are identified by the array of handles previously returned by DSS_IFACE_IOCTL_QOS_REQUEST_EX or individually requested by DSS_IFACE_IOCTL_QOS_REQUEST. This operation releases the flows associated with the QoS instances and makes them available for other QoS requests. All handles passed into the IOCTL must be valid. If any handle is invalid, the entire IOCTL fails, no QoS instances are released, and the error DS_EINVAL is returned to the application. The application should then call the DSS_IFACE_IOCTL_QOS_GET_STATUS IOCTL on each handle to ensure its validity. If that IOCTL returns the error DS_EINVAL, the handle is invalid and should not be used. The application may then resubmit the DSS_IFACE_IOCTL_QOS_RELEASE_EX IOCTL with the valid handles. This IOCTL needs to perform asynchronous operation and, hence, a return value of SUCCESS means only that the QoS request has been submitted. The final outcome is later indicated by a QoS event as described in Chapter 6. The application must ensure that num_handles is not larger than the number of handles actually stored in the array. This IOCTL currently supports no more than ten bundled releases and returns EINVAL if a greater number is set as num_handles. This IOCTL requires dss_iface_ioctl_qos_release_ex_type as a parameter. For CDMA targets, this operation can be performed even in a QoS-unaware system. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_SUSPEND

This operation suspends the currently granted QoS (identified by the handle) and, thus, releases all associated QoS resource reservations while the QoS session state is still maintained in both the mobile and the network. The QoS instance is identified by the handle previously returned by DSS_IFACE_IOCTL_QOS_REQUEST. The application will no longer be able to send data using this QoS instance until it has been reactivated. Any attempts to transmit data on a SUSPENDED QoS instance will result in data being routed over the default flow bearing no QoS guarantees. This IOCTL must be completed asynchronously and will cause sockets to return a QOS_AVAILABLE_DEACTIVATED_EV to the application once the operation has completed. If the QoS instance is already suspended, QOS_AVAILABLE_DEACTIVATED_EV will be generated immediately (possibly even before the IOCTL returns). An invalid QoS handle will return the error DS_EINVAL to the applications. On CDMA targets this operation will fail on a QoS-unaware system and will return DS_EQOSUNAWARE. This IOCTL requires dss_iface_ioctl_qos_suspend_type as a parameter. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_SUSPEND_EX

This IOCTL attempts to transition *n* QoS instances to the suspended state. The outcome of the states for each QoS instance, however, could ultimately be different. The QoS instances are identified by the handles previously returned by DSS_IFACE_IOCTL_QOS_REQUEST or DSS_IFACE_IOCTL_QOS_REQUEST_EX. This operation suspends the currently granted QoS (identified by the handle) and thus releases all associated QoS resource reservations while the QoS session state is still maintained in both the mobile and the network. The application is no longer able to send data using this QoS instance until it has been reactivated. Any attempts to transmit data on a SUSPENDED QoS instance results in data being routed over the default flow bearing no QoS guarantees. All handles passed into the IOCTL must be valid. If any handle is invalid, the entire IOCTL fails, no QOS instances are suspended, and the error DS_EINVAL is returned to the application. The application should then call the DSS_IFACE_IOCTL_QOS_GET_STATUS IOCTL on each handle. If the IOCTL returns the error DS_EINVAL, the handle is invalid and should not be used. The application may then resubmit the DSS_IFACE_IOCTL_QOS_SUSPEND_EX IOCTL with the valid handles. This IOCTL needs to perform asynchronous operation and hence a return value of SUCCESS only means that the QoS request has been submitted. The final outcome is later indicated by individual QoS events as described in Chapter 6. If the QoS instance is already suspended, the corresponding event is generated right away (possibly even before the IOCTL returns). The application must ensure that the num_handles value is not larger than the number of handles actually stored in the array. This IOCTL currently supports no more than ten bundled suspends, and will return EINVAL if a greater number is set as num_handles. This IOCTL requires dss_iface_ioctl_qos_suspend_ex_type as a parameter. On CDMA targets, this operation fails on a QoS-unaware system and returns DS_EQOSUNAWARE. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_RESUME

This IOCTL activates a suspended QoS instance. Activating a suspended QoS instance implies allocating QoS as specified by the corresponding QoS handle. The QoS is allocated based upon the QoS parameters last specified for this QoS instance. The application will once again be able to transmit data over this QoS instance once it has received the QOS_AVAILABLE_EV or QOS_AVAILABLE_MODIFIED_EV, which is returned once the asynchronous operation has completed. If the QoS instance is already activated, these events will be generated immediately (possibly before the IOCTL returns). An invalid QoS handle will error DS_EINVAL to the applications. This IOCTL will cause sockets to generate a QOS_AVAILABLE_EV/QOS_AVAILABLE_MODIFIED_EV to the application if the operation is successful, and QOS_AVAILABLE_DEACTIVATED_EV if the operation fails. On CDMA targets this operation will fail on a QoS unaware system and will return DS_EQOSUNAWARE. This IOCTL requires dss_iface_ioctl_qos_resume_type as a parameter. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_RESUME_EX

This IOCTL attempts to activate *n* suspended QoS instances. The outcome of the states for each QoS instance, however, could ultimately be different, since the network may not be able to allocate sufficient resources at that time. Activating a suspended QoS instance implies allocating QoS as specified by the corresponding QoS handle. The QoS is allocated based upon the QoS parameters last specified for this QOS instance. The application will once again be able to transmit data over this QoS instance once it is activated. This IOCTL needs to perform asynchronous operation and hence a return value of SUCCESS only means that the QoS request has been submitted. The final outcome is later indicated by *n* QoS events (one per QoS instance). If the QoS instance is already activated, a corresponding event is generated right away (possibly before the IOCTL returns). All handles passed into the IOCTL must be valid. If any handle is invalid the entire IOCTL fails, no QoS instances are resumed, and the error DS_EINVAL is returned to the application. The application should then call the DSS_IFACE_IOCTL_QOS_GET_STATUS IOCTL on each handle. If the IOCTL returns the error DS_EINVAL, the handle is invalid and should not be used. The application may then resubmit the DSS_IFACE_IOCTL_QOS_RESUME_EX IOCTL with the valid handles. The application must ensure that the num_handles value is not larger than the number of handles actually stored in the array. This IOCTL currently supports no more than ten bundled resumes, and returns EINVAL if a number greater than that is set as num_handles. This IOCTL requires dss_iface_ioctl_qos_resume_ex_type as a parameter. On CDMA targets, this operation fails on a QoS unaware system and returns DS_EQOSUNAWARE. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY

This operation is used to modify the previously requested QoS on a best effort service flow (i.e., the primary link of an interface), in Rx, Tx, or both directions, or to request a new QoS in a direction for which a QoS was not requested previously. As it is invalid to install filters on a best-effort service flow, only flow specifications can be modified. This IOCTL needs to perform asynchronous operation and, hence, a return value of SUCCESS means only that the request to modify QoS has been submitted. The final outcome is later indicated by a QoS event. This IOCTL is supported on UMTS targets only. The data type used as argval is dss_iface_ioctl_primary_qos_modify_type. For further details on the QoS architecture and usage model, see Chapter 6.

4.2.4.2.1 DSS_IFACE_IOCTL_QOS_GET_STATUS

This IOCTL can be used to query the status of a given QoS instance. The IOCTL will return one of the following status values: QOS_ACTIVATING, QOS_AVAILABLE, QOS_SUSPENDING, QOS_DEACTIVATED, QOS_RELEASING, or QOS_UNAVAILABLE. If a handle is passed that no longer points to a valid QoS instance, DS_EINVAL will be returned to the application and status field will be invalid. This IOCTL requires dss_iface_ioctl_qos_get_status_type as a parameter. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_GET_FLOW_SPEC

This operation has been deprecated. The semantics of this operation are identical to those of DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC.

DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC

This operation returns the current QoS flow specification in use by a secondary link of the interface. The QoS instance is identified by the handle previously returned by DSS_IFACE_IOCTL_QOS_REQUEST or in a Network Initiated QoS CB notification following registration via DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST or in the response of of DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST if a matching QoS was available. The IOCTL returns the flow specification for both receive and transmit directions. The flow specification currently in use can be different from the flow specification previously requested using DSS_IFACE_IOCTL_QOS_REQUEST since the network may not be able to service the desired QoS flow specification. The data type used as argval is dss_iface_ioctl_qos_get_flow_spec_type. For further details on the QoS architecture and usage model see Chapter 6.

DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC2

This operation returns the QoS flow specification in use by a secondary link of the interface along with its QoS status. The QoS instance is identified by the handle previously returned by DSS_IFACE_IOCTL_QOS_REQUEST or in a Network Initiated QoS CB notification following registration via DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST or in the response of of DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST if a matching QoS was available. The IOCTL returns the flow specification for both receive and transmit directions. The returned flow specification can be different from the flow specification previously requested using DSS_IFACE_IOCTL_QOS_REQUEST since the network may not be able to service the desired QoS flow specification. The data type used as argval is dss_iface_ioctl_qos_get_granted_flow_ spec2_type. The returned flow specification represents a currently granted flow specification only when QoS status is QOS_AVAILABLE. In all other cases, the returned flow specification is just a hint of what was granted earlier. For further details on the QoS architecture and usage model see Chapter 6.

DSS_IFACE_IOCTL_PRIMARY_QOS_GET_GRANTED_FLOW_SPEC

This operation returns the QoS flow specification in use by a primary link of the interface. The IOCTL returns the flow specification for both Rx and Tx directions. The granted flow specification can be different from the flow specification previously requested using PDP profiles or DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY, since the network may not be able to service the desired QoS flow specification. The data type used as argval is dss_iface_ioctl_primary_qos_get_granted_flow_spec_type. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_QOS_GET_GRANTED_FILTER_SPEC

This operation returns the QoS filter specification in use by a secondary link of the interface along with its QoS status. The QoS instance is identified by the handle previously returned by DSS_IFACE_IOCTL_QOS_REQUEST or in a Network Initiated QoS CB notification following registration via DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST or in the response of of DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST if a matching QoS was available. The IOCTL returns the filter specification for both receive and transmit directions. The returned filter specification can be different from the filter specification previously requested using DSS_IFACE_IOCTL_
QOS_REQUEST or DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST due to network constraints and configuration. The data type used as argval is dss_iface_ioctl_qos_get_granted_ filter_spec_type. For further details on the QoS architecture and usage model refer to Chapter 6 of this document.

DSS_IFACE_IOCTL_GET_NETWORK_SUPPORTED_QOS_PROFILES

This operation returns the current QoS profiles supported by the network. This IOCTL is currently only supported by the CDMA 1X/DO interface. The data type used as argval is dss_iface_ioctl_get_network_supported_qos_profiles_type. DSS_IFACE_IOCTL_707_NETWORK_SUPPORTED_QOS_PROFILES_CHANGED_EV is posted if the supported QoS profile set is changed by the network. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_ON_QOS_AWARE_SYSTEM

This operation is used to query whether the AT is in a QoS aware system. This IOCTL is a best guess estimate of whether QoS is available on the system. It is possible that in spite of the IOCTL returning that QoS is available, the actual QoS request may fail. The data type used as argval is dss_iface_ioctl_on_qos_aware_system_type. Currently this is supported only for CDMA targets. For further details on the QoS architecture and usage model, see Chapter 6.

DSS_IFACE_IOCTL_GO_ACTIVE

This operation is used to activate a dormant primary or secondary physical link. A primary link is the one that carries best-effort traffic, while a secondary link is the one that carries traffic with specific QoS guarantees. Activating a dormant link implies establishing a traffic channel and other appropriate air interface channels. A physical link is identified by the iface_id argument. The iface_id corresponding to a primary link can be obtained by using the API dss_get_iface_id(). The iface_id corresponding to a secondary link can be obtained from the QoS handle using the API dss_get_iface_id_by_qos_handle() as described in Section 4.2.3. This IOCTL can generate both PHYS_LINK_DOWN_EV and PHYS_LINK_UP_EV, depending upon whether the physical link can be successfully activated or not. If the physical link is already activated, the operation completes successfully but the PHYS_LINK_UP_EV is not generated right away (it would have been generated when the physical link was actually activated). This IOCTL does not require any parameters.

DSS_IFACE_IOCTL_GO_DORMANT

This operation forces a primary or a secondary physical link to go dormant. A primary link is the one that carries best-effort traffic, while a secondary link is the one that carries traffic with specific QoS guarantees. Dormancy on a link implies the releasing traffic channel and other air interface channels to free up appropriate radio resources. A physical link is identified by the

iface_id argument. The iface_id corresponding to a primary link can be obtained by using the API dss_get_iface_id(). The iface_id corresponding to a secondary link can be obtained from the QoS handle using the API dss_get_iface_id_by_qos_handle as described in Section 4.2.3. The operation is completed asynchronously and a PHYS_LINK_DOWN_EV is generated on completion. If the physical link is already dormant, operation completes successfully but the PHYS_LINK_DOWN_EV is not generated (it would have been generated when the physical link actually went dormant). Applications must explicitly register for physical link events using DSS_IFACE_IOCTL_REG_EVENT_CB with the iface_id. This IOCTL does not require any parameters.

DSS_IFACE_IOCTL_GO_NULL

This operation is used to go NULL, i.e., to tear down the interface. If FEATURE_JCDMA_2 is enabled, the call will also pass back a reason for the network teardown. The data type used as argval is dss_iface_ioctl_null_arg_type.

DSS_IFACE_IOCTL_IPCP_DNS_OPT

This operation is used to enable or disable DNS negotiation during ipcp configuration. The data type used as argval is dss_iface_ioctl_ipcp_dns_opt_type.

DSS_IFACE_IOCTL_MT_REG_CB

This operation is used by an application to register for Mobile Terminated (MT) call notification delivered through the event DSS_IFACE_IOCTL_MT_REQUEST_EV. The data type used as argval is dss_iface_ioctl_mt_reg_cb_type. This structure includes an application-specified callback of type dss_iface_ioctl_event_cb, a unique handle assigned to the application by the interface handler, a void * user_data_ptr (which is returned to the application as an argument to the callback), and a network handle.

This IOCTL registers the application to receive the event DSS_IFACE_IOCTL_MT_REQUEST_EV and returns a unique handle, which can be subsequently used for deregistration of the event. When an MT packet data call is received that matches the network policy specified by the application, the application is notified with the event DSS_IFACE_IOCTL_MT_REQUEST_EV. On receiving this event, the application should initiate the packet call by calling dssnet_start(). All applications that register for the MT call are notified of the event provided their network policy matches the MT call profile.

NOTE: The application is not allowed to change its network policy while it is registered for the DSS_IFACE_IOCTL_MT_REQUEST_EV. It must first deregister this event by calling DSS_IFACE_IOCTL_MT_DEREG_CB so that it can change the network policy and then reregister for the event. The new policy will be considered in determining whether the application should be notified with a subsequent MT call.

DSS_IFACE_IOCTL_MT_DEREG_CB

This operation is used by an application to deregister from the mobile terminated call notification delivered through DSS_IFACE_IOCTL_MT_REQUEST_EV. The data type used as argval is dss_iface_ioctl_mt_dereg_cb_type, which consists of the handle previously returned to the application during MT call registration and the nethandle. The handle must be valid and identifies the MT call instance to be deregistered.

DSS_IFACE_IOCTL_MCAST_JOIN

This IOCTL will allow an application to join a multicast group over a specific multicast interface. The data type used as argval is dss_iface_ioctl_mcast_join_type. This data type takes seven different parameters; the first two are the Multicast IP address and nonzero port number to join. The void * mcast_param_ptr contains any specific multicast data needed by a particular technology. Currently, this is not used and should be set to NULL. This structure also includes an application-specified callback of type dss_iface_ioctl_event_cb, a void * user_data_ptr (which is returned to the application as an argument to the callback), a unique handle assigned to the application by the interface handler, and a network handle. See Chapter 7 for more information on multicast support.

DSS_IFACE_IOCTL_MCAST_LEAVE

This operation will cause an application to leave a multicast group. The data type used as argval is dss_iface_ioctl_mcast_leave_type, which consists of the nethandle and the handle previously assigned during the MCAST_JOIN IOCTL call. The handle must be valid and represents the multicast group joined. Upon successful completion of this call, the application will no longer be registered for the multicast group or multicast events on that interface. See Chapter 7 for more information on multicast support.

DSS_IFACE_IOCTL_MCAST_JOIN_EX

This IOCTL allows an application to bundle multiple flows in a single IOCTL call to join a multicast group over a specific multicast interface. The data type used as argval is dss_iface_ioctl_mcast_join_ex_type. This data type takes nine different parameters. The first two are arrays of the multicast IP address and nonzero port numbers to join. The array void * mcast_param_ptr contains any specific multicast data needed by a particular technology per flow. Currently, this is not used and the contents of the array should be set to NULL. This structure also includes an application-specified callback of type dss_iface_ioctl_event_cb, a void * user_data_ptr (which is returned to the application as an argument to the callback). An array mcast_request_flags of type dss_iface_ioctl_mcast_join_ex_req_flags_enum_type is included to specify the requested operation for each flow, setup, and trigger registration for that flow (DSS_IFACE_IOCTL_MCAST_REG_SETUP_ALLOWED) or to set up the flow to be registered later (DSS_IFACE_IOCTL_MCAST_REG_SETUP_NOT_ALLOWED). An unsigned integer num_flows is included to specify the number of bundled flows in each IOCTL call and can have a maximum value up to DSS_IFACE_MAX_MCAST_FLOWS. The structure contains an array of unique handles assigned to each of the bundled flows of the application by the interface handler if the IOCTL call succeeds. Otherwise, if the IOCTL operation fails for any one instance, all instances are cleaned up and no multicast handle is returned. A network handle dss_nethandle is included in the structure to identify the application. See Chapter 7 for more information on multicast support.

DSS_IFACE_IOCTL_MCAST_LEAVE_EX

This IOCTL allows an application to specify bundled multiple flows to leave a multicast group. The data type used as argval is dss_iface_ioctl_mcast_leave_ex_type, which consists of the nethandle, an array of multicast handles (representing the multicast flows) previously assigned during the MCAST_JOIN/MCAST_JOIN_EX IOCTL call, and the number of flows bundled that can have a maximum value of DSS_IFACE_MAX_MCAST_FLOWS. The handles must be valid and should represent the multicast group joined. Upon successful completion of this call, the application will no longer be registered for the multicast group or multicast events on that interface for the requested flows. See Chapter 7 for more information on multicast support.

DSS_IFACE_IOCTL_MCAST_REGISTER_EX

This IOCTL allows an application to trigger multicast group registration for multiple flows previously setup using MCAST_JOIN_EX IOCTL call (with the DSS_IFACE_IOCTL_MCAST_REG_SETUP_NOT_ALLOWED operation). The data type used as argval is dss_iface_ioctl_mcast_register_ex_type, which consists of the nethandle, an array of previously asigned multicast handles and the number of flows bundled, which can have a maximum value of DSS_IFACE_MAX_MCAST_FLOWS. The handles must be valid and should represent the multicast group they were setup for. Upon successful completion of this IOCTL, the application is registered for multicast group or multicast events on that interface for the requested flows. See Chapter 7 for more information on multicast support.

The 707 (1X) specific IOCTLs

These operations are exclusively supported by the 707 (1X) interface.

DSS_IFACE_IOCTL_707_GET_MDR

This operation is used to get the 1X QCMDR value. The data type used as argval is dss_iface_ioctl_707_mdr_type.

DSS_IFACE_IOCTL_707_SET_MDR

This operation is used to set the 1X QCMDR value. The data type used as argval is dss_iface_ioctl_707_mdr_type.

DSS_IFACE_IOCTL_707_GET_DORM_TIMER

This operation is used to get the 1X dorm timer value. The data type used as argval is dss_iface_ioctl_707_dorm_timer.

DSS_IFACE_IOCTL_707_SET_DORM_TIMER

This operation is used to set the 1X dorm timer value. The data type used as argval is dss_iface_ioctl_707_dorm_timer.

DSS_IFACE_IOCTL_707_GET_RLP_ALL_CURR_NAK

This operation is used to get the 1X RLP curr NAK policy information. The data type used as argval is dss_iface_ioctl_707_rlp_opt_type.

DSS_IFACE_IOCTL_707_SET_RLP_ALL_CURR_NAK

This operation is used to set the 1X RLP curr NAK policy information. The data type used as argval is dss_iface_ioctl_707_rlp_opt_type.

DSS_IFACE_IOCTL_707_GET_RLP_ALL_DEF_NAK

This operation is used to get the 1X RLP default NAK information. The data type used as argval is dss_iface_ioctl_707_rlp_opt_type.

DSS_IFACE_IOCTL_707_SET_RLP_ALL_DEF_NAK

This operation is used to set the 1X RLP default NAK information. The data type used as argval is dss_iface_ioctl_707_rlp_opt_type.

DSS_IFACE_IOCTL_707_GET_RLP_ALL_NEG_NAK

This operation is used to get the 1X RLP negotiated NAK information. The data type used as argval is dss_iface_ioctl_707_rlp_opt_type.

DSS_IFACE_IOCTL_707_GET_RLP_QOS_NA_PRI

This operation is used to get the 1X QoS nonassured priority type. The data type used as argval is dss_iface_ioctl_707_qos_na_pri_type.

DSS_IFACE_IOCTL_707_SET_RLP_QOS_NA_PRI

This operation is used to set the 1X QoS nonassured priority type. The data type used as argval is dss_iface_ioctl_707_qos_na_pri_type.

DSS_IFACE_IOCTL_707_SDB_SUPPORT_QUERY

This operation is used to query the 1X interface about SDB support information. The data type used as argval is dss_iface_ioctl_707_sdb_support_query_type.

DSS_IFACE_IOCTL_707_ENABLE_HOLDDOWN

This operation is used to enable or disable the 1X holddown timer. The data type used as argval is dss_iface_ioctl_enable_holddown_type.

DSS_IFACE_IOCTL_707_ENABLE_HDR_HPT_MODE

This option is used to enable HDR high priority mode on 1x EV-DO.In 1xEV-DO HPT mode, EV-DO traffic may not be interrupted by the IS-2000 modem for IS-2000 paging or acquisition. The IS-2000 modem will use the secondary receiver for IS-2000 paging rather than the primary receiver so that EV-DO traffic may continue in parallel. In the event of IS-2000 system loss, acquisition will be delayed until the EV-DO traffic session terminates. The only IS-2000 operations that may interrupt an EV-DO HPT session are IS-2000 access attempts.

DSS_IFACE_IOCTL_707_ENABLE_HDR_REV0_RATE_INTERIA

This operation is used to set the mode for operating VT on a Rev 0 HDR system. This will end up sending a GAUP message to the AN for changing the RTCMAC rate transition parameters. This may also change the DPA RLP parameters.

DSS_IFACE_IOCTL_707_HDR_GET_RMAC3_INFO

This operation is used to get the MAC layer information for the flow associated with the specified interface. This IOCTL is valid only when RMAC3 is negotiated and the current system is DO Rev A. The data type used as argval for this operation is dss_iface_ioctl_707_hdr_rmac3_info_type.

DSS_IFACE_IOCTL_707_GET_TX_STATUS

This operation is used to get the status of the data transmission for the flow associated with the specified interface. This IOCTL returns the transmission status since the last time this IOCTL was called. If there is an error in transmission of data because of the following two reasons, the IOCTL will return FALSE; otherwise it will return TRUE. Reasons for transmission error:

- Physical Layer Transmission error (M-ARQ)

  or

- Data has become stale (by remaining in transmit watermark for too long) and thus discarded before a transmission is even attempted.

The data type used as argval for this operation is dss_iface_ioctl_707_tx_status_type.

DSS_IFACE_IOCTL_707_GET_SESSION_TIMER

This IOCTL is used to get the value of the PPP session close timer of the associated interface.

DSS_IFACE_IOCTL_707_SET_SESSION_TIMER

This IOCTL is used to set the value of the PPP session close timer of the associated interface.

DSS_IFACE_IOCTL_707_GET_INACTIVITY_TIMER

This IOCTL is used to get the value of the inactivity timer of the associated QoS instance. The data type used as argval for this operation is dss_iface_ioctl_707_inactivity_timer_type. See Section 0 for more details on the inactivity timer.

DSS_IFACE_IOCTL_707_SET_INACTIVITY_TIMER

This IOCTL is used to set the value of the inactivity timer of the associated QoS instance. If data is not transmitted on this QoS instance for the specified inactivity timer value, the traffic channel is released in order to conserve resources. The IOCTL should be called with an interface ID that can be obtained by calling dss_get_iface_id_by_qos_handle() after QoS has been requested. This ensures that the IOCTL is called on the correct QoS instance. Although the IOCTL name implies that there is a timer per every QoS instance, there is only a timer per traffic channel. Since more than one QoS instance can be multiplexed on to the same traffic channel, the largest value of all inactivity timer values is used to start the inactivity timer. A systemwide default value could be set using DSS_IFACE_IOCTL-707-SET_SESSION_TIMER, else a value of 0, i.e., no dormancy, is used by default. The data type used as argval for this operation is dss_iface_ioctl_707_inactivity_timer_type.

DSS_IFACE_IOCTL_707_GET_ HYSTERESIS_ACT_TIMER

This IOCTL is used to get the value of the Hysteresis Activation Timer (HAT). This will retrieve the current status of the HAT and, if the timer is active, the IOCTL will retrieve the remaining value of the timer.

A returned value of -1 indicates that HAT is inactive; any other value returned indicates HAT is active and the value represents the time remaining.

The data type used as argval for this operation is dss_iface_ioctl_707_hat_type.

DSS_IFACE_IOCTL_707_SET_ HYSTERESIS_ACT_TIMER

This IOCTL is used to set the value of the HAT. If it is unlikely that no further data will be exchanged with the base station, then applications will use this IOCTL to set or reset the HAT. When applications call this IOCTL, the AMSS sets the HAT to the new value. If the HAT is already active, then it will be stopped and started again, but with the new value.

Once applications get the status of HAT through GET IOCTL (Section 0), the applications can choose to introduce the hysteresis sooner by restarting the HAT with a value smaller than the remaining value returned by GET IOCTL.

The data type used as argval for this operation is dss_iface_ioctl_707_hat_type.

UMTS specific IOCTLs

These IOCTLs are supported only by the UMTS interface.

DSS_IFACE_IOCTL_3GPP_FORCE_PDN_DISCONNECT

This IOCTL is used to bring down the PDN if this is the last PDN that is currently up

BCMCS-specific IOCTLs

These IOCTLs are supported only by the BCMCS interface.

DSS_IFACE_IOCTL_BCMCS_DB_UPDATE

This operation is used to update the BCMCS database with a new flow/ip mapping. The data type used as argval is dss_iface_ioctl_bcmcs_db_update_type. The required parameters that must be populated for the flow/ip to be correctly mapped are: multicast_ip, port, flow_id, flow_id_len, zone, and overwrite. The overwrite variable specifies whether or not to overwrite an entry in the database that matches the multicast_ip, port, and zone.

DSS_IFACE_IOCTL_BCMCS_ENABLE_HANDOFF_REG

This IOCTL allows an application to enable handoff optimizations while registering multiple multicast flows. The data type used as argval is dss_iface_ioctl_bcmcs_enable_handoff_reg_type. This data type takes three different parameters as follows:

- mcast_addr_info – Multicast IP address and port

  The first is the structure that stores the multicast IP address and non zero port number to join.

- num_mcast_addr – Number of multicast addresses requested

  The second is the number of multicast addresses requested

- dss_nethandle – DSS Net handle (app_id)

  The third is the unique handle assigned to the application by the interface handler

DSS_IFACE_IOCTL_BCMCS_BOM_CACHING_SETUP

This IOCTL enables BOM caching at the time that a QChat™ call comes up. The data type used as argval is dss_iface_ioctl_bcmcs_bom_caching_setup_type. This data type takes two different parameters as follows:

- bom_caching_setup – BOM caching setup

The first parameter is the BOM caching setup enum which tells whether to enable or disable strict BOM caching.

- bom_cache_timeout – BOM cache period timeout

  The second parameter is the BOM cache timeout parameter.

4.2.4.3 MBMS specific IOCTLs

These operations are applicable only on the Multimedia Broadast/Multicast Services (MBMS) supporting interfaces.

DSS_IFACE_IOCTL_MBMS_MCAST_CONTEXT_ACTIVATE

This operation is used to activate an MBMS context for the multicast IP address and the PDP profile ID passed in as input. The data type used as argval is dss_iface_ioctl_mbms_mcast_context_act_type. An application is required to populate the ip_addr, profile_id, event_cb, user_data_ptr, and nethandle. The output parameter is the handle that the application should use when it wants to deactivate the MBMS context.

The application is asynchronously notified of the success or failure of this IOCTL in the callback function event_cb. The event returned for a success case is DSS_IFACE_IOCTL_MBMS_CONTEXT_ACT_SUCCESS_EV and for the failure case, the event returned is DSS_IFACE_IOCTL_MBMS_CONTEXT_ACT_FAILURE_EV.

DSS_IFACE_IOCTL_MBMS_MCAST_CONTEXT_DEACTIVATE

This operation is used to deactivate an MBMS multicast context that has been activated previously using the DSS_IFACE_IOCTL_MBMS_MCAST_CONTEXT_ACTIVATE IOCTL call. The data type used as argval is dss_iface_ioctl_mbms_mcast_context_deact_type. The application is required to populate the nethandle and the handle it received during MBMS context activation.

The application is asynchronously notified of the success or failure of this IOCTL in the callback function event_cb. The event returned for a success case is DSS_IFACE_IOCTL_MBMS_CONTEXT_DEACT_SUCCESS_EV, and for the failure case, the event returned is DSS_IFACE_IOCTL_MBMS_CONTEXT_DEACT_FAILURE_EV.

Example

This operation retrieves the IPV4 address of the interface being used by an application.

```
dss_iface_id_type iface_id;
dss_iface_ioctl_ipv4_addr_type ip_addr;
sint15 dss_errno;

if ((iface_id = dss_get_iface_id(my_nethandle)) == DSS_IFACE_INVALID_ID)
{
  MSG_ERROR("Failed to get the iface id for nethandle=%d", my_nethandle, 0,
0);
  return;
}
```

```
        dss_iface_ioctl(iface_id, DSS_IFACE_IOCTL_GET_IPV4_ADDR, &ip_addr,
        &ds_errno);
```

4.2.4.4 SoftAP specific IOCTLs

These IOCTLs are only supported for the SoftAP feature. SoftAP should be ON on the device before invoking these IOCTLs.

DSS_IFACE_IOCTL_ENABLE_FIREWALL

This IOCTL is used to enable the firewall. The data type used as argval for this IOCTL is dss_iface_ioctl_enable_firewall_type. This data type has two parameters. The first parameter is the network handle. The second parameter is a boolean to indicate whether to allow or drop the incoming packets that match the firewall rules. TRUE implies that only the packets matching the firewall rules are allowed and rest of the traffic is dropped. FALSE implies that the packets matching the firewall rules are dropped and the rest of the traffic is allowed to pass.

DSS_IFACE_IOCTL_DISABLE_FIREWALL

This IOCTL is used to disable the firewall. The data type used as argval for this IOCTL is dss_iface_ioctl_disable_firewall_type. This data type has one parameter. The only parameter is the network handle.

DSS_IFACE_IOCTL_ADD_FIREWALL_RULE

This IOCTL is used to add the firewall rules. The data type used as argval for this IOCTL is dss_iface_ioctl_add_firewall_rule_type. This data type has three parameters. The first parameter is the network handle. The second parameter is the filter specification that forms the firewall rule. The last parameter is the handle that is returned to the caller on successful creation of the firewall rule. The user can use this handle in subsequent requests to delete or read the firewall rule. Note that the firewall should have been enabled before any firewall rule becomes effective.

DSS_IFACE_IOCTL_DELETE_FIREWALL_RULE

This IOCTL is used to delete a firewall rule. The data type used as argval for this IOCTL is dss_iface_ioctl_delete_firewall_rule_type. This data type has two parameters. The first parameter is the network handle. The second parameter is the handle that the caller should have acquired when it invoked DSS_IFACE_IOCTL_ADD_FIREWALL_RULE to add the firewall rule.

DSS_IFACE_IOCTL_GET_FIREWALL_RULE

This IOCTL is used to read a firewall rule using the handle that it acquired when the firewall rule was added. The data type used as argval for this IOCTL is dss_iface_ioctl_get_firewall_rule_type. This data type has three parameters. The first parameter is the network handle. The second parameter is the handle that the caller should have acquired when it invoked DSS_IFACE_IOCTL_ADD_FIREWALL_RULE to add the firewall rule. The last parameter is the IP filter specification forming the firewall rule which is populated by the IOCTL handler.

DSS_IFACE_IOCTL_GET_FIREWALL_TABLE

This IOCTL is used to read all the firewall rules currently set. The rules should have been added previously using DSS_IFACE_IOCTL_ADD_FIREWALL_RULE IOCTL.

The data type used as argval for this IOCTL is dss_iface_ioctl_get_firewall_table_type. This data type has four parameters. The first parameter is the network handle populated by the caller. The second parameter is the number of filters requested by the user. The third parameter is the place holder for the number of filters requested. The memory for this is allocated by the caller and the values are populated by the IOCTL handler with the firewall rules. The last parameter is the value that the IOCTL handler populates indicating the available number of filters.

It is possible to first retrieve the available number of filters be passing a NULL buffer in the fltr_spec_arr parameter to the IOCTL. Once the available number of filters is known then the user can allocate memory for that number of filters and call the IOCTL again to retrieve all the firewall rules.

DSS_IFACE_IOCTL_ADD_STATIC_NAT_ENTRY

This IOCTL is used to add a static NAT entry that is needed to set port forwarding rules at the NAT. The data type used as argval for this operation is dss_iface_ioctl_add_static_nat_entry_type. This data type takes five parameters. The first parameter is the network handle populated by the caller.The second parameter is the private IP address in network byte order. The third parameter is the global port. The fourth parameter is the Private port and the last parameter is the protocol (TCP/UDP/ICMP/ESP). Private IP address and the Private Port fields forms the Destination <IP:Port> pair for any packet received on the global port.

DSS_IFACE_IOCTL_DELETE_STATIC_NAT_ENTRY

This IOCTL is used to delete a static NAT entry. The data type used as argval for this operation is dss_iface_ioctl_delete_static_nat_entry_type. This data type has the same parameters that are listed above in DSS_IFACE_IOCTL_ADD_STATIC_NAT_ENTRY.

DSS_IFACE_IOCTL_GET_STATIC_NAT_ENTRY

This IOCTL is used to get all the static NAT entries that are set using the IOCTL DSS_IFACE_IOCTL_ADD_STATIC_NAT_ENTRY. The data type used as argval for this operation is dss_iface_ioctl_get_static_nat_entry_type. This data type has four parameters. The first parameter is the network handle populated by the caller. The second parameter is the place holder for the static NAT entries. The memory for this is allocated by the caller and the IOCTL handler populates the values. The third parameter is the number of static entries that the caller has allocated memory for. The last parameter is the total number of static NAT entries present and is populated by the IOCLT handler.

It is possible to first retrieve the available number of NAT entries be passing a NULL buffer to the IOCTL. Once the available number of entries is known then the user can allocate memory for that number of entries and call the IOCTL again to retrieve all the static NAT entries.

DSS_IFACE_IOCTL_GET_DYNAMIC_NAT_ENTRY_TIMEOUT

This IOCTL is used to get NAT entry timeout value. The data type used as argval for this operation is dss_iface_ioctl_get_dynamic_nat_entry_timeout_type. This data type has two parameters. The first parameter is the network handle populated by the caller. The second parameter is the timeout value specified in seconds.

DSS_IFACE_IOCTL_SET_DYNAMIC_NAT_ENTRY_TIMEOUT

This IOCTL is used to set the NAT entry timeout value. The data type used as argval for this operation is dss_iface_ioctl_set_dynamic_nat_entry_timeout_type. This data type has the same parameters that are listed above in DSS_IFACE_IOCTL_GET_DYNAMIC_NAT_ENTRY_TIMEOUT.

DSS_IFACE_IOCTL_SET_NAT_IPSEC_VPN_PASS_THROUGH

This IOCTL is used to set the NAT IPSec VPN PassThrough mode (TRUE or FALSE). The data type used as argval for this operation is dss_iface_ioctl_nat_ipsec_vpn_pass_through_type. This data type has two parameters. The first parameter is the network handle populated by the caller. The second parameter is the boolean value indicating if the VPN Pass-through should be enabled (TRUE) or disabled (FALSE).

DSS_IFACE_IOCTL_GET_NAT_IPSEC_VPN_PASS_THROUGH

This IOCTL is used to get the NAT IPSec VPN PassThrough mode (TRUE or FALSE). The data type used as argval for this operation is dss_iface_ioctl_nat_ipsec_vpn_pass_through_type. This data type has the same parameters that are listed above in DSS_IFACE_IOCTL_SET_NAT_IPSEC_VPN_PASS_THROUGH.

DSS_IFACE_IOCTL_DHCP_ARP_CACHE_UPDATE

This IOCTL is used to update the ARP cache with DHCP client's IP/MAC per the lease offered by the server. The data type used as argval for this operation is dss_iface_ioctl_dhcp_arp_cache_update_type. The first parameter is the IP address of the client. The second parameter is the hardware address of the client. The last parameter is the hardware address length.

DSS_IFACE_IOCTL_DHCP_ARP_CACHE_CLEAR

This IOCTL is used to clear an entry from the ARP cache. The data type used as argval for this operation is dss_iface_ioctl_dhcp_arp_cache_clear_type. This data type has the same parameters that are listed above in DSS_IFACE_IOCTL_DHCP_ARP_CACHE_UPDATE.

DSS_IFACE_IOCTL_DHCP_SERVER_GET_DEVICE_INFO

This IOCTL is used to get information on the devices that are connected to DHCP at any given point of time. The data type used as argval for this operation is dss_iface_ioctl_dhcp_get_device_info_type. This data type has three parameters. The first parameter is the place holder for the connected devices information. The memory for this is allocated by the caller and the IOCTL handler populates the values. The second parameter is the number of connected devices for which the caller has allocated memory. The last parameter is the total number of connected devices present and is populated by the IOCLT handler. This IOCTL should be called on WLAN interface.

DSS_IFACE_IOCTL_ADD_DMZ

This IOCTL is used to add the DMZ (DeMilitarized zone) entry. The data type used as argval for this IOCTL is dss_iface_ioctl_dmz_type. This data type has two parameters. The first parameter is the network handle. The second parameter is the IP address of the client that would act as a DMZ.

DSS_IFACE_IOCTL_GET_DMZ

This IOCTL is used to get the DMZ (De-Militarized Zone) entry that has been added previously using the DSS_IFACE_IOCTL_ADD_DMZ IOCTL. The data type used as argval for this IOCTL is dss_iface_ioctl_dmz_type. This data type has two parameters. The first parameter is the network handle. The second parameter is the IP address of the client that would act as a DMZ.

DSS_IFACE_IOCTL_DELETE_DMZ

This IOCTL is used to delete the DMZ (DeMilitarized zone) entry that has been added previously using DSS_IFACE_IOCTL_ADD_DMZ IOCTL. The data type used as argval for this IOCTL is dss_iface_ioctl_delete_dmz_type. This data type has one parameter which is the network handle.

DSS_IFACE_IOCTL_GET_NAT_PUBLIC_IP_ADDR

This IOCTL is used to get the public IP address of the NAT interface. The data type used as argval for this IOCTL is dss_iface_ioctl_nat_public_ip_addr_type. This data type has one parameter. The parameter is the placeholder for public IP address of the NAT interface. This IOCTL currently supports getting IPV4 address of the NAT interface.

DSS_IFACE_IOCTL_GET_IFACE_STATS

This IOCTL is used to get the statistics of the interface. The data type used as argval for this operation is dss_iface_ioctl_iface_stats_type.

DSS_IFACE_IOCTL_RESET_IFACE_STATS

This IOCTL is used to reset the statistics of the interface. No data type is used as argval and this can be passed as NULL.

DSS_IFACE_IOCTL_GET_NAT_L2TP_VPN_PASS_THROUGH

This IOCTL is used to get the NAT L2TP VPN PassThrough mode (enabled/disabled). The data type used as argval for this operation is dss_iface_ioctl_nat_l2tp_vpn_pass_through_type. This data type has the same parameters that are listed above in DSS_IFACE_IOCTL_SET_NAT_L2TP_VPN_PASS_THROUGH.

DSS_IFACE_IOCTL_SET_NAT_L2TP_VPN_PASS_THROUGH

This IOCTL is used to enable/disable the NAT L2TP VPN PassThrough mode. The data type used as argval for this operation is dss_iface_ioctl_nat_l2tp_vpn_pass_through_type. This data type has two parameters. The first parameter is the network handle populated by the caller. The second parameter is the boolean value indicating if the VPN PassThrough should be enabled (TRUE) or disabled (FALSE).

DSS_IFACE_IOCTL_GET_NAT_PPTP_VPN_PASS_THROUGH

This IOCTL is used to get the NAT PPTP VPN PassThrough mode (enabled/disabled). The data type used as argval for this operation is dss_iface_ioctl_nat_pptp_vpn_pass_through_type. This data type has the same parameters that are listed above in DSS_IFACE_IOCTL_SET_NAT_PPTP_VPN_PASS_THROUGH.

DSS_IFACE_IOCTL_SET_NAT_PPTP_VPN_PASS_THROUGH

This IOCTL is used to enable/disable the NAT PPTP VPN PassThrough mode. The data type used as argval for this operation is dss_iface_ioctl_nat_pptp_vpn_pass_through_type. This data type has two parameters. The first parameter is the network handle populated by the caller. The second parameter is the boolean value indicating if the VPN PassThrough should be enabled (TRUE) or disabled (FALSE).

Example

This operation retrieves the IPV4 address of the interface being used by an application.

```
dss_iface_id_type iface_id;
dss_iface_ioctl_ipv4_addr_type ip_addr;
sint15 dss_errno;

if ((iface_id = dss_get_iface_id(my_nethandle)) == DSS_IFACE_INVALID_ID)
{
  MSG_ERROR("Failed to get the iface id for nethandle=%d", my_nethandle, 0, 0);
  return;
}
dss_iface_ioctl(iface_id, DSS_IFACE_IOCTL_GET_IPV4_ADDR, &ip_addr, &ds_errno);
```

5 Sockets Programming for Multiple PDP Profiles

The UMTS air interface provides support for multiple Packet Data Protocol (PDP) contexts. Each PDP context maps to a unique network interface. Each PDP context is assigned its own IP address during PDP context establishment. If more than one PDP context is active simultaneously, the mobile effectively becomes a multi-homed device, i.e., the mobile would have multiple network interfaces. This section of the document addresses the programming model for multiple PDP contexts. The applications running on the phone can explicitly bring up and bind to specific PDP context. Up to three such contexts can be set up. Note that if an external PDP context activation exists using a laptop for example, the number of profiles that the application can start is limited to two.

The list of parameters required to activate a PDP context are grouped together and called PDP profile parameters. The mobile stores *N* of these PDP profiles in permanent storage (the file system). The sockets API Programming model provides an interface for applications to specify which PDP context they would like to activate and bind to by specifying the number of the PDP profile to be used. A network interface is bound to the application when the application calls dsnet_start(). All sockets created using the dss_socket() function will use the interface that is bound to the application as their transmit interface.

5.1 Socket application using default PDP profile on UMTS

An application that wants to use the UMTS interface needs to set the interface name (DSS_IFACE_UMTS) explicitly in the policy structure. Alternatively, if the application does not specify the interface name and the mobile is currently camped in a UMTS/GPRS system, the mobile would setup a call using the UMTS interface. An application has the option to use the default PDP profile specified in the default policy, if it is not particular about the PDP profile used. Below is sample code for a socket application that uses the UMTS interface with the default PDP profile. If the application does not specify the profile number or an explicit policy and if the mobile is camped on UMTS or GPRS, a default PDP profile is activated. This default PDP profile can be set using UI menus and the value is preserved across power cycles.

Example 1

```
/*-----------------------------------------------------------------
In this example, application explicitly sets the interface name to
DSS_UMTS_IFACE and uses the default PDP profile.
-----------------------------------------------------------------*/
dss_net_policy_info my_net_policy;
sint15 errno;
sint15 nethandle;
/*-----------------------------------------------------------------
Initialize the network policy and specify the interface name in the
policy structure. Allocate a network handle.
-----------------------------------------------------------------*/
dss_init_net_policy_info(&my_net_policy);
my_net_policy->iface.kind      = DSS_IFACE_NAME;
my_net_policy->iface.info.name = DSS_IFACE_UMTS;
nethandle = dsnet_get_handle(my_net_cb, NULL, my_sock_cb,
                         NULL, &my_net_policy, &errno);

/*-----------------------------------------------------------------
Start the network. The UMTS interface with the default PDP profile will be
bound to this application during this process.
-----------------------------------------------------------------*/
dsnet_start(nethandle, &errno);

/*-----------------------------------------------------------------
Once the application is notified that the network is up via the
network callback (my_net_cb), it can use the UMTS network with the
default PDP profile to transfer data as desired.
-----------------------------------------------------------------*/
```

Example 2

```
/*-----------------------------------------------------------------
In this example, application does not explicitly set the interface
name to DSS_UMTS_IFACE. If the mobile is camped on a UMTS/GPRS system, the
default PDP context would be established.
-----------------------------------------------------------------*/
dss_net_policy_info my_net_policy;
sint15 errno;
sint15 nethandle;
/*-----------------------------------------------------------------
Initialize the network policy structure. Allocate a network handle.
-----------------------------------------------------------------*/
dss_init_net_policy_info(&my_net_policy);
nethandle = dsnet_get_handle(my_net_cb, NULL, my_sock_cb,
                         NULL, &my_net_policy, &errno);

/*-----------------------------------------------------------------
Start the network. The UMTS interface with the default PDP profile
will be bound to this application during this process.
-----------------------------------------------------------------*/
dsnet_start(nethandle, &errno);

/*-----------------------------------------------------------------
Once the application is notified that the network is up via the
network callback (my_net_cb), it can use the UMTS network with the default
PDP profile to transfer data as desired.
-----------------------------------------------------------------*/
```

5.2 Socket application using specific PDP profile on UMTS

An application that wants to use a specific PDP profile on the UMTS interface can explicitly choose this using the policy structure. Below is sample code for a simple socket application that uses the UMTS interface with a specific PDP profile. Even though two applications request different profile numbers, they might bind to the same interface if the profile definition (APN, PDP type, and PDP address) of the two profiles matches.

If an application wants to bind to more than one PDP profile, it has to allocate one network handle for each PDP profile. During socket creation, the network handle that is specified in dss_socket() call, determines the PDP profile that the socket uses to transmit data.

Example 3

```
/*-----------------------------------------------------------------
In this example, application explicitly sets the interface name and
also specifies the PDP profile(MY_DESIRED_PDP).
-----------------------------------------------------------------*/
dss_net_policy_info my_net_policy;
sint15 errno;
sint15 nethandle;

/*-----------------------------------------------------------------
Initialize the network policy and specify the interface name in the policy
structure.
-----------------------------------------------------------------*/
dss_init_net_policy_info(&my_net_policy);
my_net_policy.iface.kind      = DSS_IFACE_NAME;
my_net_policy.iface.info.name = DSS_IFACE_UMTS;

/*-----------------------------------------------------------------
Set the PDP profile number to the desired value (MY_DESIRED_PDP). Allocate
a network handle.
-----------------------------------------------------------------*/
my_net_policy.umts.pdp_profile_num = MY_DESIRED_PDP;
nethandle = dsnet_get_handle(my_net_cb, NULL, my_sock_cb,
                        NULL, &my_net_policy, &errno);

/*-----------------------------------------------------------------
Start the network. The UMTS interface with the MY_DESIRED_PDP profile will
be bound to this application during this process.
-----------------------------------------------------------------*/
dsnet_start(nethandle, &errno);

/*-----------------------------------------------------------------
Once the application is notified that the network is up via the network
callback (my_net_cb), it can use the UMTS network with the MY_DESIRED_PDP
profile to transfer data as desired.
-----------------------------------------------------------------*/
```

Example 4

```
/*-----------------------------------------------------------------
In this example, application wants to use two different PDP profiles
(MY_DESIRED_PDP1 and MY_DESIRED_PDP2) to transmit data.
-----------------------------------------------------------------*/
dss_net_policy_info my_net_policy1, my_net_policy2;
sint15 errno;
sint15 nethandle1, nethandle2;
sint15 sock_fd1, sock_fd2;

/*-----------------------------------------------------------------
Initialize the first network policy and set PDP profile to MY_DESIRED_PDP1.
Allocate the first network handle and start the network.
-----------------------------------------------------------------*/
dss_init_net_policy_info(&my_net_policy1);
my_net_policy1.iface.kind     = DSS_IFACE_NAME;
my_net_policy1.iface.info.name = DSS_IFACE_UMTS;
my_net_policy1.umts.pdp_profile_num = MY_DESIRED_PDP1;
nethandle1 = dsnet_get_handle(my_net_cb, NULL, my_sock_cb,
                              NULL, &my_net_policy1, &errno);
dsnet_start(nethandle1,&errno);

/*-----------------------------------------------------------------
Initialize the second network policy and set PDP profile to
MY_DESIRED_PDP2. Allocate the second network handle and start the network.
-----------------------------------------------------------------*/
dss_init_net_policy_info(&my_net_policy2);
my_net_policy2.iface.kind      = DSS_IFACE_NAME;
my_net_policy2.iface.info.name = DSS_IFACE_UMTS;
my_net_policy2.umts.pdp_profile_num = MY_DESIRED_PDP2;
nethandle2 = dsnet_get_handle(my_net_cb, NULL, my_sock_cb,
                              NULL, &my_net_policy2, &errno);
dsnet_start(nethandle2,&errno);

/*-----------------------------------------------------------------
```

```
      Create two sockets, one bound to nethandle1 and the second bound to
      nethandle2.
      ------------------------------------------------------------------*/
      sock_fd1 = dss_socket(nethandle1,AF_INET, SOCK_STREAM,IPPROTO_TCP,
         &errno);
      sock_fd2 = dss_socket(nethandle2,AF_INET, SOCK_STREAM,IPPROTO_TCP,
         &errno);
      /*------------------------------------------------------------------
      Any data transmitted using sock_fd1 would use MY_PDP_DESIRED1 and any data
      transmitted using sock_fd2 would use MY_PDP_DESIRED2
      ------------------------------------------------------------------*/
```

6 QoS Architecture and Usage Model

This chapter describes the Sockets API Programming model to use QoS, provided by underlying IP interfaces. QoS support allows each incoming and outgoing IP packet stream to be treated as per the desired quality parameters to guarantee specific application behavior.

Sockets API extensions to support QoS have been described earlier in the document. This section provides the details on QoS architecture, theory of operation, and the QoS specification block. The AMSS Sockets API currently supports QoS for IPv4 for cdma2000®, UMTS, and WLAN technologies. Only the UMTS technology, however, currently supports QoS for IPv6.

6.1 Theory of operation

In order for an application to request QoS (UE Initiated QoS), it needs to issue DSS_IFACE_IOCTL_QOS_REQUEST IOCTL with desired QoS specification. API automatically registers the application for QoS events, which are described in the next section, and returns a unique QoS handle to identify the requested QoS instance. Note that applications are not allowed to explicitly register/deregister for QoS events using DSS_IFACE_IOCTL_REG_EVENT_CB or DSS_IFACE_IOCTL_DEREG_EVENT_CB. The network may, or may not, grant the most desirable QoS or may reject the QoS request. If QoS is granted by the network, the required QoS resources are reserved in both the mobile and the network, and the application is notified with an appropriate event. Any subsequently transmitted or received data on the flow matching the requested QoS flow receives QoS guarantees permitted by the granted QoS specification.

Application may also register for notifications of Network Initiated QoS. Such registration can be made via DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST. Application may specify QoS filters specification of interest and shall only be notified if a QoS initiated by the Network matches that filter specification. When application is notified on a matching Network Initiated QoS, the CB provides a unique QoS handle to identify the Network Initiated QoS instance. The system may share the QoS instance between several applications but each application shall be provided with a unique handle to the QoS instance. With the notification to application on a matching Network Initiated QoS, the application is automatically registered for QoS events on that QoS instance.

The network may decide to modify, or revoke a granted QoS at any time, in which case the application is notified with an appropriate event. For UE initiated QoS, the application may decide to modify the requested QoS using DSS_IFACE_IOCTL_QOS_MODIFY operation. The network may accept or reject this operation and if the operation is rejected, the network may decide to keep the old QoS. For UE initiated QoS, the application may also suspend a QoS instance, i.e., during inactivity, by issuing DSS_IFACE_IOCTL_QOS_SUSPEND IOCTL. A suspended QoS instance can be resumed by the application by either issuing DSS_IFACE_IOCTL_QOS_RESUME IOCTL or transmitting data on the QoS flow, but the former approach is recommended to avoid delays in setting up QoS resources during data transmission.

Also, note that technology mode handlers may allow one or more QoS instances over a secondary link. When all QoS instances using a secondary link are suspended, a technology mode handler may choose to make the link dormant. Alternately, a secondary link can be forced dormant by using a DSS_IFACE_IOCTL_GO_DORMANT operation on the link. When the first QoS instance using a secondary link is resumed, a technology mode handler may activate the link if it happens to be dormant. For UE initiated QoS, applications can also force the secondary link active by using a DSS_IFACE_IOCTL_GO_ACTIVE operation. If only one QoS instance is allowed on a secondary link, the technology mode handler may suspend/resume a flow when the link is made dormant/active and vice versa.

In all the above cases, depending on the technology mode handler, an application may receive events on both the QoS instance and the secondary physical link even if it performed only one operation. Although an application is expected to handle this behavior, it is not recommended to rely on it.

If an application wishes to know the current state of its QoS instance it may use DSS_IFACE_IOCTL_QOS_GET_STATUS operation. An application may also monitor the state of the physical link over which the QoS is flowing by either calling DSS_IFACE_IOCTL_GET_PHYS_LINK_STATE or by explicitly registering for phys_link_events using DSS_IFACE_IOCTL_REG_EVENT_CB.

Since the network may not grant the most desirable QoS, an application may want to know the granted QoS using a DSS_IFACE_IOCTL_GET_GRANTED_FLOW_SPEC operation. When an application no longer requires QoS, it should release QoS using a DSS_IFACE_IOCTL_QOS_RELEASE IOCTL. An application is automatically deregistered with the QoS events when QoS is released by either the application or the network.

6.1.1 QoS flow architecture

A QoS instance is represented by a flow entity. There can be one or more flows per network interface. Each flow is bound to one physical link entity, and there can be multiple flows bound to the same physical link. Figure 6-1 illustrates the hierarchical relationship of interfaces, flows, and physical links.

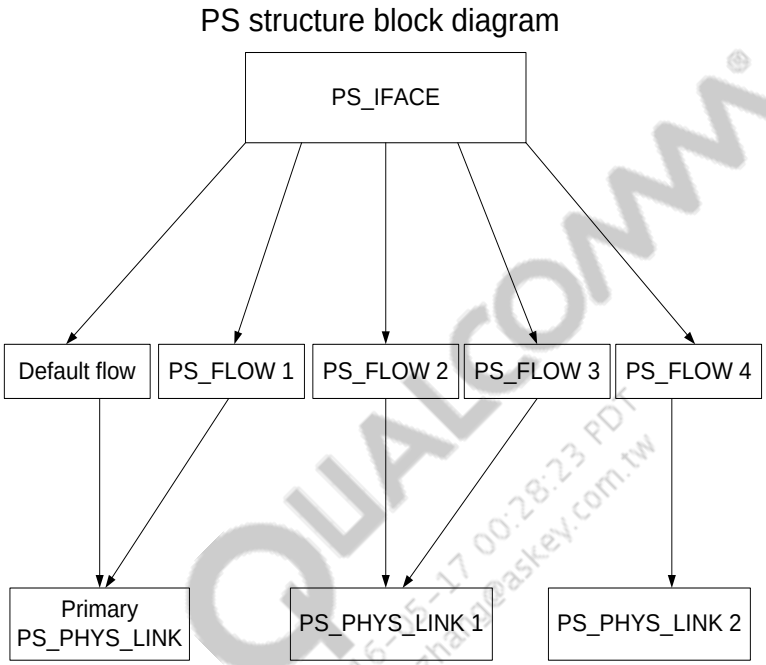


Figure 6-1 QoS flow architecture

An IP flow that does not require any specific QoS guarantees is called a default flow. Any IP traffic that is not associated with a QoS flow is part of the default flow. Other IP flows are called QoS flows and require specific QoS guarantees. The physical link that carries the default flow is called primary physical link. A primary physical link may also carry other QoS flows. All other physical links are called secondary physical links and have specific attributes to provide desired QoS to the QoS flows bound to those physical links.

6.1.2 Routing semantics

When QoS is requested (or when Network Initiates QoS), a new flow entity is created and filters are installed in the transmit path. If a transmitted packet matches any of these filters, the packet is routed over the corresponding QoS flow. If for any reason, a packet cannot be transmitted on that QoS flow, it will be routed on the default flow, unless DSS_SO_DISABLE_FLOW_FWDING socket option is set. Packets are forwarded on to default flow in the following scenarios:

- QoS flow is still being set up and is not activated yet.
- QoS flow is being released as a result of DSS_IFACE_IOCTL_QOS_RELEASE IOCTL from the application.

- The QoS flow is suspended or is being suspended. In this case, the packet is routed on the default flow and the sockets layer may try to activate the corresponding QoS flow so that subsequent packets may be routed over it. As sockets layer activating QoS flow is not always guaranteed, it is recommended that an application always calls DSS_IFACE_IOCTL_QOS_RESUME to activate the suspended flow.
- If the physical link associated with the QoS flow is dormant or in the middle of going dormant, the packet is routed on the default flow but the corresponding secondary physical link is also attempted to be activated so that subsequent packets can be routed over it. An application does not need to call DSS_IFACE_IOCTL_GO_ACTIVE to activate the dormant physical link in this case.

6.2 QoS event handling

This section describes Quality of Service (QoS) events and the way they are reported to the application.

6.2.1 Event callback

Events are reported using a callback function of type dss_iface_ioctl_event_cb, which it passes back to the application an iface_id and a QoS handle. The iface_id identifies the interface on which the event has occurred and the QoS handle identifies the QoS instance. The iface_id is the same ID that was passed down with the DSS_IFACE_IOCTL_QOS_REQUEST IOCTL or DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST IOCTL.

The Event Callback function is defined as:

```
void (*dss_iface_ioctl_event_cb) (
                dss_iface_ioctl_event_enum_type        event,
                dss_iface_ioctl_event_info_union_type  event_info,
                Void                                   *user_data,
                sint15                                 nethandle,
                dss_iface_id_type                      iface_id
                )
```

→	dss_iface_ioctl_event_cb

	→	event	The event being reported defined by dss_iface_ioctl_event_enum_type The following QoS related events have been added: ▪ DSS_IFACE_IOCTL_QOS_AVAILABLE_MODIFIED_EV –QoS is available and may or may not be the most desirable QoS; see Section 6.2.2.2 for details ▪ DSS_IFACE_IOCTL_QOS_UNAVAILABLE_EV – QoS could not be provided or previously granted QoS is unavailable now; see Section 6.2.2.3 for details ▪ DSS_IFACE_IOCTL_QOS_AVAILABLE_DEACTIVATED_ EV – QoS is suspended; see Section 6.2.2.4 for details ▪ DSS_IFACE_IOCTL_QOS_MODIFY_ACCEPTED_EV – QoS specification requested in DSS_IFACE_IOCTL_QOS_MODIFY or DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY operation is accepted by the network; see Section 6.2.2.5 for details ▪ DSS_IFACE_IOCTL_QOS_MODIFY_REJECTED_EV – QoS specification requested in DSS_IFACE_IOCTL_QOS_MODIFY or DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY operation is rejected by the network; see Section 6.2.2.6 for details ▪ DSS_IFACE_IOCTL_QOS_NET_INITIATED_AVAILABLE_EV – Network has initiated QoS which filters match the filters specification provided by application in DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST.Ssee Section 6.2.2.11 for details ▪ DSS_IFACE_IOCTL_QOS_INFO_CODE_UPDATED_EV – New information is available for a QoS instance; see Section 6.2.2.10 for details

	→	event_info	Information related to the event being reported defined by dss_iface_ioctl_event_info_union_type. Information related to QoS events is provided by a dss_iface_ioctl_qos_event_info_type member that contains two fields: handle, which was previously returned by DSS_IFACE_IOCTL_QOS_REQUEST, identifies the QoS instance and an info_code, which optionally provides extended information regarding the event being reported. The extended information is defined by dss_iface_ioctl_extended_info_code_enum_type and info_code is set to DSS_IFACE_IOCTL_EIC_NOT_SPECIFIED in case no extended information is available. The extended information code values that can be returned are: ▪ DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_MIN – Indicates the minimum value for all the information codes which are generated based on information internal to the mobile ▪ DSS_IFACE_IOCTL_EIC_QOS_INVALID_PARAMS – This information code is deprecated and is replaced by DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_INVALID_ PARAMS. ▪ DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_INVALID_ PARAMS. Invalid parameters are passed in to a QoS API and validation is failed in one of the layers on the mobile. ▪ DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_CALL_ ENDED – Call is terminated locally by the mobile ▪ DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_ERROR – An internal error has occurred during QoS-related processing ▪ DSS_IFACE_IOCTL_EIC_QOS_INSUFFICIENT_LOCAL_ RESOURCES – MS does not have sufficient resources to support desired QoS ▪ DSS_IFACE_IOCTL_EIC_QOS_TIMEDOUT_OPERATION – Operation timed out due to no response from the network. ▪ DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_UNKNOWN_ CAUSE_CODE – An internal error has occurred but corresponding information code is not defined for this error yet ▪ DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_MAX – Indicates the maximum value for all the information codes which are generated based on information internal to the mobile. All the internal information codes will be in the range [DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_MIN, DSS_IFACE_IOCTL_EIC_QOS_INTERNAL_MAX] ▪ DSS_IFACE_IOCTL_EIC_QOS_NETWORK_MIN – Indicates the minimum value for all the information codes which are generated based on information received from the network ▪ DSS_IFACE_IOCTL_EIC_QOS_NOT_SUPPORTED – QoS feature is not supported by the network ▪ DSS_IFACE_IOCTL_EIC_QOS_NOT_AVAILABLE – Requested QoS cannot be provided ▪ DSS_IFACE_IOCTL_EIC_QOS_NOT_GUARANTEED – QoS granted earlier is no longer

			▪ DSS_IFACE_IOCTL_EIC_QOS_INSUFFICIENT_NET_ RESOURCES – Network does not have enough resources to support desired QoS ▪ DSS_IFACE_IOCTL_EIC_QOS_AWARE_SYSTEM – MS is currently in a system that supports QoS ▪ DSS_IFACE_IOCTL_EIC_QOS_UNAWARE_SYSTEM – MS is currently in a system that does not support QoS. ▪ DSS_IFACE_IOCTL_EIC_QOS_REJECTED_OPERATION – Operation was rejected by the network ▪ DSS_IFACE_IOCTL_EIC_QOS_WILL_GRANT_WHEN_ QOS_RESUMED – Currently not used ▪ DSS_IFACE_IOCTL_EIC_QOS_NETWORK_CALL_ ENDED – Call is terminated by the network ▪ DSS_IFACE_IOCTL_NETWORK_SVC_NOT_AVAILABLE – Mobile is not camped on any network ▪ DSS_IFACE_IOCTL_ NETWORK_L2_LINK_RELEASED – Network released layer 2 link ▪ DSS_IFACE_IOCTL_NETWORK_L2_LINK_ REESTABLISH_REJ – Network rejected mobile's attempt to reestablish layer 2 link ▪ DSS_IFACE_IOCTL_ NETWORK_L2_LINK_REESTABLISH_IND – Network accepted mobile's attempt to re-establish layer 2 link ▪ DSS_IFACE_IOCTL_EIC_QOS_NETWORK_MAX – Indicates the maximum value for all the information codes which are generated based on information received from the network. All the network information codes will be in the range [DSS_IFACE_IOCTL_EIC_QOS_NETWORK_MIN, DSS_IFACE_IOCTL_EIC_QOS_NETWORK_MAX] ▪ Also note that handle is NULL if event is generated in response to DSS_IFACE_IOCTL_PRIMARY_QOS_ MODIFY or if the event is DSS_IFACE_IOCTL_QOS_ MODIFY_R ESULT_EV
	→	`user_data`	Application-specific data, which is previously passed into DSS_IFACE_IOCTL_QOS_REQUEST API to request QoS
	→	`nethandle`	Network handle
	→	`iface_id`	Interface handle, which is previously used to request QoS in DSS_IFACE_IOCTL_QOS_REQUEST API

6.2.2 QoS events

This section explains the semantics of QoS events in detail.

6.2.2.1 DSS_IFACE_IOCTL_QOS_AVAILABLE_EV

This event is deprecated. DSS_IFACE_IOCTL_QOS_AVAILABLE_MODIFIED_EV is generated instead.

6.2.2.2 DSS_IFACE_IOCTL_QOS_AVAILABLE_MODIFIED_EV

This event is generated under the following conditions:

- In response to DSS_IFACE_IOCTL_QOS_REQUEST or an DSS_IFACE_IOCTL_QOS_RESUME operation that has returned SUCCESS to indicate that QoS, which may or may not be the most desirable QoS, is now available and a flow has been set up to provide the granted QoS.
- When the network forcibly modifies a previously granted QoS, which, for instance, can happen during a network handoff or when the network degrades a previously granted QoS due to unavailability of resources.
- When the network activates a suspended QoS instance because of data, which matches the filters installed in the Rx direction by the application, coming in to the mobile from the network. QoS is not granted if the application receives data that does not match any filter.

In response to this event, it is recommended that the applications should query the currently granted QoS using DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC and take an appropriate action if the granted QoS is not sufficient for proper functioning of the application. If acceptable QoS is not granted, the application may choose to release the QoS instance and report an error to the user and retry the original QoS request using DSS_IFACE_IOCTL_QOS_MODIFY after waiting for a period of time, although it may still not get the requested QoS if the network does not have sufficient resources to fulfill it. For each occurrence, a reason for which this event is generated may be provided by info_code field in event_info.

6.2.2.3 DSS_IFACE_IOCTL_QOS_UNAVAILABLE_EV

This event indicates that the requested QoS could not be granted or the granted QoS is not available anymore. This event is generated:

- In response to successful completion of a DSS_IFACE_IOCTL_QOS_RELEASE operation that has returned SUCCESS previously
- if the network is unable to grant QoS and does not want to maintain a QoS session state, in response to DSS_IFACE_IOCTL_QOS_REQUEST, DSS_IFACE_IOCTL_QOS_RESUME, or DSS_IFACE_IOCTL_QOS_MODIFY operations that have returned SUCCESS previously
- When the network has forcibly revoked previously allocated QoS; this could happen due to various reasons like network running out of resources or a handoff to an older network system that does not support QoS, etc.

For each occurrence, specific reasons for which the QoS is not available may be indicated in the info_code passed with the event_info. The event implies that the flow associated with the QoS

instance has also been freed up. Note that the application is still able to exchange data over the network if the network is up, although no specific QoS can be guaranteed.

6.2.2.4 DSS_IFACE_IOCTL_QOS_AVAILABLE_DEACTIVATED_EV

This event indicates that an assigned QoS is deactivated and all the resources are released. This event is generated:

- In response to successful completion of a DSS_IFACE_IOCTL_QOS_SUSPEND operation that has previously returned SUCCESS
- If the network is unable to grant QoS but wants to maintain QoS Session state, in response to a DSS_IFACE_IOCTL_QOS_REQUEST, DSS_IFACE_IOCTL_QOS_RESUME, or DSS_IFACE_IOCTL_QOS_MODIFY operations that have previously returned SUCCESS
- When the network deactivates the QoS, i.e., due to inactivity on the flow
- When the mobile hands off to a QoS unaware system from a QoS aware system

For each occurrence, a specific reason why the QoS is deactivated may be indicated in the info_code passed with the event_info. This event implies that the QoS instance and the associated flow are still valid and configured with QoS parameters but the resource reservations no longer exist.

6.2.2.5 DSS_IFACE_IOCTL_QOS_MODIFY_ACCEPTED_EV

This event is generated in response to a DSS_IFACE_IOCTL_QOS_MODIFY or DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY operation that has previously returned SUCCESS and indicates that the network can grant the QoS specified in that operation. Since the network may or may not grant the most desirable QoS, it is recommended that the applications should query the currently granted QoS using DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC or DSS_IFACE_IOCTL_PRIMARY_QOS_GET_GRANTED_FLOW_SPEC and take an appropriate action if the granted QoS is not sufficient for proper functioning of the application. If the acceptable QoS is not granted, the application may choose to release the QoS instance and report an error to the user and retry the original QoS request using DSS_IFACE_IOCTL_QOS_MODIFY/ DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY after waiting for a period of time, although it may still not get the requested QoS if the network does not have sufficient resources to fulfill it. Any change in a QoS instance's status is indicated by another appropriate event.

6.2.2.6 DSS_IFACE_IOCTL_QOS_MODIFY_REJECTED_EV

This event is generated in response to a DSS_IFACE_IOCTL_QOS_MODIFY operation that has previously returned SUCCESS and indicates that the network cannot grant the QoS specified in that operation. AMSS continues to provide the granted QoS prior to a DSS_IFACE_IOCTL_QOS_MODIFY operation. Any change in a QoS instance's status is indicated by another appropriate event.

6.2.2.7 DSS_IFACE_IOCTL_QOS_AWARE_SYSTEM_EV

This event must be explicitly registered and deregistered by the application using the IOCTLs DSS_IFACE_IOCTL_REG_EVENT_CB and DSS_IFACE_IOCTL_DEREG_EVENT_CB.

Refer to the DSS API for the specifics on how these IOCTLs work. This event is registered on an iface and is not tied to any particular qos instance. This event will be generated immediately upon registration if the mobile is already in a QOS aware system. In addition, this event is also generated in response to a handoff to a system that supports QOS.

6.2.2.8 DSS_IFACE_IOCTL_QOS_UNAWARE_SYSTEM_EV

This event must be explicitly registered and deregistered by the application using the IOCTLs DSS_IFACE_IOCTL_REG_EVENT_CB and DSS_IFACE_IOCTL_DEREG_EVENT_CB. Please refer to the DSS API for the specifics on how these IOCTLs work. This event is registered on an iface and is not tied to any particular qos instance. This event will be generated immediately upon registration if the mobile is in a QOS unaware system. In addition, this event is also generated in response to a handoff to a system that does not support QOS.

6.2.2.9 DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY_RESULT_EV

This event must be explicitly registered and deregistered by the application using the IOCTLs DSS_IFACE_IOCTL_REG_EVENT_CB and DSS_IFACE_IOCTL_DEREG_EVENT_CB. This event is registered on an iface. This event is generated to indicate whether the QoS modification on the primary link of an iface is successful. QoS modification on the primary link can be triggered by either the application or the network.

6.2.2.10 DSS_IFACE_IOCTL_QOS_INFO_CODE_UPDATED_EV

This event is generated when new information related to a QoS instance is available. The info_code passed in event_info contains the update.

6.2.2.11 DSS_IFACE_IOCTL_QOS_NET_INITIATED_AVAILABLE_EV

This event is generated to notify the application on Network Initiated QoS, which filters specification matches the specification provided by the application in a call to DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST. Event information is provided in structure of type dss_iface_ioctl_qos_net_initiated_available_event_info_type. It contains a handle to the new QoS as well as the handle of the notification registration request provided to application on return from DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST.

6.3 QoS specification block

This section describes the QoS specification block in detail. QoS is requested using a block of information called the QoS specification block. This section also describes in detail the semantics and usage of the QoS block. The data services API allows for specifying QoS independently for each direction, Rx and Tx. The direction refers to the direction of data transfer from the MS to the network. The QoS specification consists of two parts, flow specification and filter specification. Flow specification defines the treatment required for an IP flow while the filter specification defines the IP flow itself. Each IP flow can constitute several IP streams belonging to one or more higher-layer protocols. The flow specification contains a req_flow, which defines the most suitable flow parameters, a min_req_flow, which defines the minimum required flow parameters, and an aux_flow_list_ptr, which defines auxiliary sets of flow types (in decreasing order of desirability) in between req_flow and min_req_flow.

Each interface handler may handle these different sets of flow specifications differently, but the overall model exposed to the applications is the same. An interface handler always tries to obtain the most desirable flow parameters from the network as specified by req_flow. If the most desirable flow parameters cannot be granted, the technology handler ensures that the granted flow parameters are at least as good as specified by the min_req_flow. In both cases, the application is notified with a DSS_IFACE_IOCTL_QOS_AVAILABLE_MODIFIED_EV. Otherwise, the QoS request is rejected and the application is notified with a DSS_IFACE_IOCTL_QOS_UNAVAILABLE_EV. It is invalid to specify either min_req_flow type or aux_flow_list_ptr flow types without specifying req_flow type.

Whenever an application is notified with a DSS_IFACE_IOCTL_QOS_AVAILABLE_MODIFIED_EV, it can query the currently applied flow type using the DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC.

For each direction for which the QoS is desired, a filter specification with one or more filters and at least req_flow type must be specified.

For Network Initiated QoS, flow specification is not supported as part of the application request.

The details of the QoS specification and usage semantics are described next.

6.3.1 QoS specification block definition

```
/*---------------------------------------------------------------------
Definition of high level QoS specification block. Contains parameters QoS
in Rx and Tx or both directions.
---------------------------------------------------------------------*/
typedef enum
{
  QOS_MASK_RX_FLOW,
  QOS_MASK_RX_MIN_FLOW,
  QOS_MASK_RX_AUXILIARY_FLOWS,
  QOS_MASK_TX_FLOW,
  QOS_MASK_TX_MIN_FLOW,
  QOS_MASK_TX_AUXILIARY_FLOWS,
  QOS_MODIFY_MASK_RX_FLTR_MODIFY,
  QOS_MODIFY_MASK_TX_FLTR_MODIFY
 } qos_spec_field_mask_enum_type;

typedef uint16 qos_spec_field_mask_type;

typedef struct
{
  qos_spec_field_mask_type field_mask;

  struct
  {
    struct ip_flow_spec    flow_template;/* traffic flow template */
    struct ip_filter_spec   fltr_template;/* traffic filter template*/
  } rx;

  struct
  {
    struct ip_flow_spec    flow_template;/* traffic flow template */
    struct ip_filter_spec   fltr_template;/* traffic filter template*/
  } tx;
} qos_spec_type;
```

→	qos_spec_type		
	→	field_mask	The bit mask to indicate which of the specification blocks are specified; see Section 6.3.1.1 for details
	→	rx	QoS specification for Rx direction This has two parts, flow_template and fltr_template, where flow_template defines the flow specification block (see Section 6.3.2 for details) while fltr_template defines the filter specification block (see Section 6.3.3 for details)
	→	tx	QoS specification for Tx direction This has two parts, flow_template and fltr_template, where flow_template defines the flow specification block (see Section 6.3.2 for details) while fltr_template defines the filter specification block (see Section 6.3.3 for details)

6.3.1.1 QoS specification semantics

Following are the semantics for the QoS specification block. A semantics violation will result in failure in an API in which the QoS block is passed in.

6.3.1.1.1 Semantics for DSS_IFACE_IOCTL_QOS_REQUEST

QoS specified in a DSS_IFACE_IOCTL_QOS_REQUEST operation shall follow the following sematics:

1. For each direction for which the QoS is requested, a flow specification with the req_flow type shall be specified, which is the most suitable flow type, and the corresponding bit in field_mask shall be set, i.e., QOS_MASK_RX_FLOW shall be set to request QoS in the Rx direction and the QOS_MASK_TX_FLOW shall be set to request QoS in the Tx direction.
2. For each direction for which the QoS is requested, min_req_flow may be specified, which is the minimum required flow type.
   a. It is invalid to specify min_req_flow type without specifying req_flow type.
   b. If specified, a corresponding bit in field_mask shall be set, i.e., QOS_MASK_RX_MIN_FLOW shall be set to request minimum QoS in the Rx direction, and QOS_MASK_TX_MIN_FLOW shall be set to request minimum QoS in the Tx direction.
3. For each direction for which the QoS is requested, auxiliary flow types may be specified by setting the num_auxr_flows to the number of auxiliary flows and aux_flow_list_ptr to the list of such flows.
   a. It is invalid to specify auxiliary flow types without specifying req_flow type.
   b. If specified, a corresponding bit in field_mask shall be set, i.e., QOS_MASK_RX_AUXILIARY_FLOWS shall be set to request auxiliary QoS in the Rx direction and QOS_MASK_TX_AUXILIARY_FLOWS shall be set to request auxiliary QoS in the Tx direction.
   c. Some network technologies (like 3GPP-based) may ignore auxiliary flows.
   d. These flow types shall be between the req_flow and min_req_flow in terms of preference.

4. For each direction in which the QoS is requested, a filter spec containing one or more filters, belonging to the same IP version as that of iface on which QoS is requested, shall be specified.
5. It is invalid to set any arbitrary bit in field_mask that is not described in this section.

6.3.1.1.2 Semantics for DSS_IFACE_IOCTL_QOS_MODIFY

QoS specified in a DSS_IFACE_IOCTL_QOS_MODIFY operation shall follow the following sematics:

1. Using this IOCTL, QoS can be modified for either Rx, Tx or both directions or a new QoS can be requested for a direction for which it was not previously requested. Either of the parameters, flow specification or filter specification, or both in each direction, can be modified for the specified QoS instance.
2. The QoS specification is divided into four blocks: Rx flow specification, Rx filter specification, Tx flow specification, and Tx filter specification. Any of these blocks can be modified. The only restriction is that if a block is modified, a new QoS specification must specify that block in full, i.e., if the application requested two filters earlier in a direction and wants to add one more filter for that direction, it has to specify all three filters in the new QoS specification. If it specifies only the filter it wants to add, AMSS replaces the two originally installed filters with the new filter.
3. In order to modify, i.e., add, change, or delete a flow specification in a direction, aacorresponding bit in field_mask shall be set, i.e., QOS_MASK_RX_FLOW shall be set to modify QoS in the Rx direction, and QOS_MASK_TX_FLOW shall be set to modify QoS in the Tx direction.
    a. One can delete an existing flow specification in a direction by setting a corresponding bit in the field_mask and by setting field_mask of req_flow type to IPFLOW_MASK_NONE.
    b. If the application wants to add or change a flow specification in a direction, rules listed in Section 6.3.1.1.1 shall apply.
4. A Filter specification can be modified in a direction by setting a corresponding bit in field_mask. “Modify” means that filters were installed in that direction previously and that the application either wants to change the number of filters or change parameters of filters, i.e., QOS_MODIFY_MASK_RX_FLTR_MODIFY shall be set to modify a filter specification in Rx direction and QOS_MODIFY_MASK_TX_FLTR_MODIFY shall be set to modify filter specification in the Tx direction.
5. If a block is not modified, i.e., none of the corresponding bits for that block are set in field_mask, it is fetched from the previously requested QoS.
6. It is invalid to set any arbitrary bit in field_mask that is not described in this section.

6.3.2 Flow specification

This section describes the flow specification parameters and their usage semantics. All parameters may not be supported by every network technology. Hence, if such parameters are requested and the underlying network technology does not support it, those parameters will be ignored. Similarly, some parameters may be supported by only a particular network technology, and other network technologies will ignore them.

Flow specification block definition

```
/*-----------------------------------------------------------------------
Definition of flow specification parameters which define a flow treatment.
-----------------------------------------------------------------------*/
typedef enum
{
  IPFLOW_MASK_NONE,
  IPFLOW_MASK_TRF_CLASS,
  IPFLOW_MASK_DATA_RATE,
  IPFLOW_MASK_LATENCY,
  IPFLOW_MASK_LATENCY_VAR,
  IPFLOW_MASK_PKT_ERR_RATE,
  IPFLOW_MASK_MIN_POLICED_PKT_SIZE,
  IPFLOW_MASK_MAX_ALLOWED_PKT_SIZE,
  IPFLOW_MASK_UMTS_RES_BER,
  IPFLOW_MASK_UMTS_TRF_PRI,
  IPFLOW_MASK_CDMA_PROFILE_ID,
  IPFLOW_MASK_WLAN_USER_PRI,
  IPFLOW_MASK_WLAN_MIN_SERVICE_INTERVAL,
  IPFLOW_MASK_WLAN_MAX_SERVICE_INTERVAL,
  IPFLOW_MASK_WLAN_INACTIVITY_INTERVAL,
IPFLOW_MASK_NOMINAL_SDU_SIZE,
IPFLOW_MASK_UMTS_IM_CN_FLAG,
IPFLOW_MASK_UMTS_SIG_IND
} ipflow_field_mask_enum_type;

typedef uint32 ipflow_field_mask_type;

typedef enum
{
  DATA_RATE_FORMAT_MIN_MAX_TYPE,
  DATA_RATE_FORMAT_TOKEN_BUCKET_TYPE
} ipflow_data_rate_format_type;

typedef enum
{
```

```
  IP_TRF_CLASS_CONVERSATIONAL,
  IP_TRF_CLASS_STREAMING,
  IP_TRF_CLASS_INTERACTIVE,
  IP_TRF_CLASS_BACKGROUND
} ip_traffic_class_enum_type;

typedef enum
{
  UMTS_RES_BIT_ERR_RATE1,        /* 5*10-2 */
  UMTS_RES_BIT_ERR_RATE2,        /* 1*10-2 */
  UMTS_RES_BIT_ERR_RATE3,        /* 5*10-3 */
  UMTS_RES_BIT_ERR_RATE4,        /* 4*10-3 */
  UMTS_RES_BIT_ERR_RATE5,        /* 1*10-3 */
  UMTS_RES_BIT_ERR_RATE6,        /* 1*10-4 */
  UMTS_RES_BIT_ERR_RATE7,        /* 1*10-5 */
  UMTS_RES_BIT_ERR_RATE8,        /* 1*10-6 */
  UMTS_RES_BIT_ERR_RATE9,        /* 6*10-8 */
  UMTS_RES_BIT_ERR_RATE10        /* Reserved */
} umts_residual_ber_enum_type;

typedef enum
{
  UMTS_TRF_HANDLING_PRI1,            /* Priority Level 1 */
  UMTS_TRF_HANDLING_PRI2,            /* Priority Level 2 */
  UMTS_TRF_HANDLING_PRI3             /* Priority Level 3 */
} umts_trf_handling_pri_enum_type;

typedef uint16 cdma_flow_spec_profile_id_type;
typedef uint8  cdma_flow_priority_type;


typedef enum
{
  WLAN_USER_PRI_BEST_EFFORT,
  WLAN_USER_PRI_BACKGROUND,
  WLAN_USER_PRI_RESERVED,
  WLAN_USER_PRI_EXCELLENT_EFFORT,
  WLAN_USER_PRI_CONTROLLED_LOAD,
  WLAN_USER_PRI_VIDEO,
  WLAN_USER_PRI_VOICE,
  WLAN_USER_PRI_NETWORK_CONTROL
} wlan_user_pri_enum_type;

typedef struct
```

```
{
  ipflow_field_mask_type          field_mask;
  ipflow_field_mask_type          err_mask;

  ip_traffic_class_enum_type      trf_class;          /* TC */

  struct
  {
    ipflow_data_rate_format_type  format_type;

    union
    {
      struct
      {
        uint32      max_rate;          /* bps, Rmax */
        uint32      guaranteed_rate;   /* bps, Rg */
      } min_max;

      struct
      {
        uint32      peak_rate;         /* bps, Rp */
        uint32      token_rate;        /* bps, Rb */
        uint32      size;              /* bytes, B */
      } token_bucket;
    } format;
  } data_rate;

  uint32        latency;            /* msec, L */
  uint32        latency_var;        /* msec, Lvar */

  struct                            /* ratio, Eper = m * 10 ** (-p) */
  {
    uint16      multiplier;         // Multiplication factor, m
    uint16      exponent;           // -ve of Base 10 exponent, p
  } pkt_err_rate;

  uint32      min_policed_pkt_size;   /* bytes, Smin */
  uint32      max_allowed_pkt_size;   /* bytes, Smax */

  struct
  {
    boolean  is_fixed;
    uint32   size;
  } nominal_sdu_size;
```

```
  /* 3GPP only QoS parameters */
  struct
  {
    umts_residual_ber_enum_type         res_ber;        /* Erber */
    umts_trf_handling_pri_enum_type     trf_pri;        /* Ptrf  */
    boolean                             im_cn_flag;
    boolean                             sig_ind;
  } umts_params;

  /* 3GPP2 only QoS parameters */
  struct
  {
    cdma_flow_spec_profile_id_type  profile_id;             /* I */
    cdma_flow_priority_type         flow_priority;
  } cdma_params;

  /* WLAN only QOS parameters */
  struct
  {
    wlan_user_pri_enum_type     user_priority;        /* User Priority */
    uint32                      min_service_interval; /* In micro sec  */
    uint32                      max_service_interval; /* In micro sec  */
    uint32                      inactivity_interval;  /* In micro sec  */
  } wlan_params;
} ip_flow_type;

typedef struct
{
  ip_flow_type      req_flow;       /* Required flow type           */
  ip_flow_type      min_req_flow;   /* Minimum required flow type   */

  /* Support for auxiliary profile specifications */
  uint8             num_aux_flows;
  ip_flow_type    * aux_flow_list_ptr;
} ip_flow_spec_type;
```

1

→	ip_flow_spec_type		
	→	req_flow	Flow parameters required for best behavior of the application
	→	min_req_flow	Minimum flow parameters required for the functioning of the application
	→	num_aux_flows	Number of specified auxiliary flow specifications
	→	aux_flow_list_ptr	Pointer to an array of auxiliary flow specifications; these flow specifications are sorted in decreasing order of preference and has low priority than req_flow and high priority than min_req_flow

→	ip_flow_type		
	→	field_mask	The bit mask specified by the caller to indicate which of the flow parameters are specified If a bit is set, the value of the corresponding flow parameter is used from the structure. If a bit is not set, a default value of the corresponding field is assumed, which may be dependent upon the network technology type. The bit mask is of type ipflow_field_mask_type and the following bits are defined: ▪ IPFLOW_MASK_TRF_CLASS – Corresponds to trf_class ▪ IPFLOW_MASK_DATA_RATE – Corresponds to data_rate ▪ IPFLOW_MASK_LATENCY – Corresponds to latency ▪ IPFLOW_MASK_LATENCY_VAR – Corresponds to latency_var ▪ IPFLOW_MASK_PKT_ERR_RATE – Corresponds to pkt_err_rate ▪ IPFLOW_MASK_MIN_POLICED_PKT_SIZE – Corresponds to min_policied_pkt_size ▪ IPFLOW_MASK_MAX_ALLOWED_PKT_SIZE – Corresponds to max_allowed_pkt_size ▪ IPFLOW_MASK_UMTS_RES_BER – Corresponds to res_ber in UMTS-specific parameters ▪ IPFLOW_MASK_UMTS_TRF_PRI – Corresponds to trf_pri in UMTS-specific parameters ▪ IPFLOW_MASK_CDMA_PROFILE_ID – Corresponds to profile_id in CDMA-specific parameters
	←	err_mask	This bit mask is returned to the caller to indicate errors in the specified flow parameters. If a bit is set, the value of the corresponding flow parameter is erroneously specified, e.g., it could be out of the allowed range. A clear bit implies that either the corresponding flow parameter was not specified, i.e., the bit was not set in the field_mask, or the specified flow parameter value is correct. The bit mask is of type ipflow_field_mask_type, which is described above. Callers should only pay attention to this field if an error is returned to the caller by the API in which ip_flow_type structure is passed in as a parameter. If the API returned success to the caller, this field is not valid and must be ignored.
	→	trf_class	Overall traffic characteristics exhibited by the IP flow; this parameter is of type ip_traffic_class_enum_type and has the following values defined: ▪ IP_TRF_CLASS_CONVERSATIONAL ▪ IP_TRF_CLASS_STREAMING ▪ IP_TRF_CLASS_INTERACTIVE ▪ IP_TRF_CLASS_BACKGROUND See Section 6.3.2.2 for further details.

	→	data_rate	Required rate for data transmission over the network; this parameter can be specified in multiple formats to provide flexibility to the applications. The format is indicated by format_type which is of type ipflow_data_rate_format_type and defines the following possible formats: ▪ DATA_RATE_FORMAT_MIN_MAX_TYPE – Data rate is specified by min_max which contains the parameters max_rate and guaranteed_rate ▪ DATA_RATE_FORMAT_TOKEN_BUCKET_TYPE – Data rate is specified by token_bucket which contains the parameters peak_rate, token_rate, and size See Section 6.3.2.2 for further details.
	→	latency	Required delay limits for transferring IP packets through the wireless link; see Section 6.3.2.2 for further details
	→	latency_var	Required delay variation limits for transferring IP packets through the wireless link; see Section 6.3.2.2 for further details
	→	pkt_err_rate	Required error rate limits for the IP packets while transferring them over wireless link; this parameter is specified using a combination of two fields, multiplier and exponent; see Section 6.3.2.2 for further details
	→	min_policed_pkt_size	Minimum IP packet size policed for QoS guarantees; see Section 6.3.2.2 for further details
	→	max_allowed_pkt_size	Maximum IP packet size that is allowed to be transferred through the QoS link; see Section 6.3.2.2 for further details
	→	nominal_sdu_size	Typical packet size used by application each time it writes to the socket; see Section 6.3.2.2 for further details

1

	→	umts_params	UMTS-specific parameters – These parameters are processed only by UMTS interfaces and are ignored by other technology interfaces. The following UMTS-specific parameters are currently defined: ▪ Residual Bit Error Rate – res_ber ▪ Traffic Handling Prioirty – trf_pri ▪ IMS Signalling Context – im_cn_flag ▪ High Priority Data – sig_ind res_ber is of type umts_residual_ber_enum_type and has the following values defined: ▪ UMTS_RES_BIT_ERR_RATE1 – Residual BER of 5*10-2 ▪ UMTS_RES_BIT_ERR_RATE2 – Residual BER of 1*10-2 ▪ UMTS_RES_BIT_ERR_RATE3 – Residual BER of 5*10-3 ▪ UMTS_RES_BIT_ERR_RATE4 – Residual BER of 4*10-3 ▪ UMTS_RES_BIT_ERR_RATE5 – Residual BER of 1*10-3 ▪ UMTS_RES_BIT_ERR_RATE6 – Residual BER of 1*10-4 ▪ UMTS_RES_BIT_ERR_RATE7 – Residual BER of 1*10-5 ▪ UMTS_RES_BIT_ERR_RATE8 – Residual BER of 1*10-6 ▪ UMTS_RES_BIT_ERR_RATE9 – Residual BER of 6*10-8 ▪ UMTS_RES_BIT_ERR_RATE10 – Reserved value trf_pri is of type umts_trf_handling_pri_enum_type and has the following values define: ▪ UMTS_TRF_HANDLING_PRI1 – Relative traffic handling priority 1 ▪ UMTS_TRF_HANDLING_PRI2 – Relative traffic handling priority 2 ▪ UMTS_TRF_HANDLING_PRI3 – Relative traffic handling priority 3 See Section 6.3.2.2 for further details.
	→	cdma_params	CDMA-specific parameters – These parameters are processed only by CDMA interfaces and are ignored by other technology interfaces. The following CDMA-specific parameters are currently defined: ▪ Profile Id – profile_id ▪ Flow priority – flow_priority profile_id is of type cdma_flow_spec_profile_id_type; currently, there are no profile ID values defined. These values will be defined in the future. Flow priority is of type cdma_flow_priority_type. See Section 6.3.2.2 for further details.

1

	→	wlan_params	WLAN-specific parameters – These parameters are processed only by WLAN interfaces and are ignored by other technology interfaces. The following WLAN-specific parameters are currently defined: ▪ User priority – user_priority ▪ Minimum service interval – min_service_interval ▪ Maximum service interval – max_service_interval ▪ Inactivity interval – inactivity_interval user_priority is of type wlan_user_pri_enum_type and has the following values defined: ▪ WLAN_USER_PRI_BEST_EFFORT ▪ WLAN_USER_PRI_BACKGROUND ▪ WLAN_USER_PRI_RESERVED ▪ WLAN_USER_PRI_EXCELLENT_EFFORT ▪ WLAN_USER_PRI_CONTROLLED_LOAD ▪ WLAN_USER_PRI_VIDEO ▪ WLAN_USER_PRI_VOICE ▪ WLAN_USER_PRI_NETWORK_CONTROL See Section 6.3.2.2 for further details.

6.3.2.1 Flow specification semantics

Following are the semantics for the flow specification block:

1. A bit in field_mask shall be set for each parameter specified. If a bit is not set, a default value for the corresponding parameter is used, which may be dependent upon the network technology handler.
2. If one or more parameters are incorrrectly specified, the bits in err_mask indicate those parameters.
3. If a parameter consisting of multiple subfields is specified, each subfield shall also be correctly specified. Such parameters are:
   a. min_max
   b. token_bucket
   c. pkt_err_rate

6.3.2.2 Flow specification parameters

Traffic class

This parameter describes the overall class of traffic to which an IP flow belongs. This parameter provides hints to the network about traffic characteristics and helps it set up the physical channels appropriately.

IP_TRF_CLASS_CONVERSATIONAL – Conversation traffic is characterized by a low transfer time because of the conversational nature of the traffic and limited delay variation of the end-to-end flow to preserve the time relation (variation) between information entities of the stream. An example of an application bearing such traffic is voice or videoconferencing.

Fundamental characteristics for QoS are:

- Preserve time relation (variation) between information entities of the stream, low Lvar
- Stringent and low delay (stringent and low L)

IP_TRF_CLASS_STREAMING – Streaming traffic is characterized by a limited delay variation of the end-to-end flow to preserve the time relation (variation) between information entities of the stream. The highest acceptable delay variation is application dependent. An example of an application bearing such traffic is videostreaming.

Fundamental characteristics for QoS are:

- Preserve time relation (variation) between information entities of the stream (low Lvar)

IP_TRF_CLASS_INTERACTIVE – Interactive traffic is characterized by the request response pattern of the end-user, and round-trip-delay (RTD) time is therefore one of the key attributes. Another characteristic is that the content of the packets shall be transferred transparently. An example of an application bearing such traffic is web browsing.

Fundamental characteristics for QoS are:

- Request response pattern
- Low RTD (low L)
- Preserve payload content (low Eper)

IP_TRF_CLASS_BACKGROUND – Background traffic is more or less delivery-time insensitive, although it requires that the content of the packets are transferred transparently. An example of an application bearing such traffic is background email download.

Fundamental characteristics for QoS are:

- The destination is not expecting the data within a certain time (no specific limit on L)
- Preserve payload content (low Eper)

If the corresponding bit in the field mask is set, this parameter is considered valid only if trf_class is one of the defined enum values.

Data rate

This parameter defines the rate at which the data is transmitted over the network. Data rate can be specified in one of two ways, a set of maximum rate and guaranteed rate values, or in the token bucket parameter style. Token bucket parameters include a peak rate, token rate, and token bucket size, and they more closely represent a packet-switched behavior with support for packet bursts. On the other hand, the former method more closely represents a circuit-switched behavior with a fixed bandwidth guaranteed over a certain time duration with no burst support. A network technology may support one of the other methods to specify data rate and, hence, each representation may require a conversion from one form to another depending upon the underlying network technology being used. The converted form may not be able to provide an exact data rate pattern as requested. Various subparameters are:

- max_rate – Maximum required data rate (in bits per second)
- guaranteed_rate – Minimum guaranteed data rate (in bits per second)
- peak_rate – Maximum rate at which the data can be transmitted when token bucket is full (in bits per second)
- token_rate – Rate at which the tokens will be put in the token bucket (in bits per second); a token is required to be present in the bucket to send a byte of data
- size – Maximum tokens that can be accumulated at any instance (in bytes); controls the size of the burst that is allowed at any given time

If the corresponding bit in the field mask is set, this parameter is considered valid only if:

- max_rate and guaranteed_rate are greater than zero and if max_rate is greater than or equal to the guaranteed_rate when min_max style is used
- peak_rate, token_rate, and size are greater than zero if the token bucket style is used

Latency

This parameter defines the maximum delay (in milliseconds) that can be tolerated by an IP packet during the transfer through the wireless link. The delay is defined as the time from a request to transfer the packet on the MS to the time when it is received by the next hop IP node across a wireless link and vice versa. If the corresponding bit in the field mask is set, this parameter is considered valid only if the latency is greater than zero.

Latency variance

This parameter defines the difference between the maximum and minimum latency (in milliseconds) that can be tolerated by an IP packet during the transfer through the wireless link. This is essentially the maximum tolerated delay jitter. This parameter is not supported by the UMTS network and hence is ignored if the underlying network is UMTS. All possible values including zero are valid for this parameter.

Packet error rate

This parameter defines the maximum packet error rate that can be tolerated by an IP flow. The error rate is defined as a ratio of packets received in error over a wireless link to total packets transmitted over the wireless link and is specified by a combination of two parameters, a multiplier m and a negative base 10 exponent. So the packet error rate is defined as:

E = m * 10 ** (-p)

The interface may use an alternate packet error rate value nearest to the one specified, depending upon the network technology. All possible values including zero are valid for this parameter.

Minimum policed packet size

This parameter defines the minimum packet size (in bytes) that will be policed for QoS guarantees. Any IP packets that are smaller than the minimum specified policed size may not receive the requested QoS. This parameter is not supported by the UMTS network and hence is ignored if the underlying network is UMTS. All possible values including zero are valid for this parameter.

Maximum allowed packet size

This parameter defines the maximum packet size (in bytes) allowed in the IP flow. The parameter is used for QoS policing and admission control. Any IP packets that are greater in size than the maximum allowed packet size are not queued for transmission over the QoS link and may get discarded. If the corresponding bit in the field mask is set, this parameter is considered valid only if max_allowed_pkt_size is greater than zero.

Nominal SDU size

This parameter specifies the typical packet size used by the application while writing data to the socket. If the subparameter is fixed is TRUE, then the application always uses a constant packet size.

If the corresponding bit in the field mask is set, this parameter is considered valid only if:

- The subparameter size is greater than zero.
- The subparameter size is equal to max_allowed_pkt_size when max_allowed_pkt_size is also specified in the flow specification and if the subparameter is_fixed is TRUE.
- The subparameter size is less than or equal to max_allowed_pkt_size when max_allowed_pkt_size is also specified in the flow specification and if the subparameter is_fixed is FALSE.

Residual bit error ratio

This parameter indicates the undetected bit error ratio for each IP flow in the delivered packets. For equal error protection, only one value is needed. If no error detection is requested for a subflow, the residual bit error ratio indicates the bit error ratio in that subflow of the delivered SDUs. If the corresponding bit in the field mask is set, this parameter is considered valid only if res_ber is one of the defined enum values.

This parameter is supported by 3GPP networks only and is ignored by all other network technologies.

Traffic handling priority

This parameter defines the relative priority between various subflows of an IP flow; it also allows the network to differentiate between bearer qualities and schedule traffic accordingly. If the corresponding bit in the field mask is set, this parameter is considered valid only if trf_pri is one of the defined enum values.

This parameter is supported by 3GPP networks only and is ignored by all other network technologies.

Profile ID

This parameter defines a set of standard/well-known flow specification parameters for specific applications. Flow specification is hence called a profile and is indicated by a profile ID. If this parameter is specified, all other flow specification parameters are ignored and the profile ID is used to request QoS from the network. Essentially, a profile ID maps to a set of predefined parameters serving a specific purpose, e.g., a profile ID for VoIP will define suitable parameters necessary for the proper functioning of various VoIP applications.

This parameter is supported by 3GPP2 networks only and is ignored by all other network technologies. Applications that require a seamless behavior on other network technologies should also specify other flow specification parameters in addition to a profile ID. Otherwise, if only a profile ID is specified and the underlying network technology is something other than 3GPP2, then the interface may not be able to provide any specific QoS. For best results, the flow specification parameter values specified should correspond to the values defined by the profile ID. This parameter is also mandatory for 3GPP2 networks and QoS specification is considered as invalid by 3GPP2 networks if this parameter is not specified.

Flow priority

This parameter defines relative priority among QoS flows. The value could be used by the network to provide admission control and resource allocation.

This parameter is supported by 3GPP2 networks only and is ignored by all other network technologies.

3GPP2 allows a range of 0 to 15 for this field and a low value has a low priority. This field is not mandatory and if not specified, a default value of 15 is used. An error is returned if a value greater than 15 is used by an application.

User priority

This parameter defines the priority of the packets sent by the application. This information can be used by the routers in packet scheduling. If the corresponding bit in the field mask is set, this parameter is considered valid only if user_priority is one of the defined enum values.

This parameter is supported by WLAN networks only and is ignored by all other network technologies.

Minimum service interval

This parameter defines the minimum interval, in microseconds, between the start of two service periods (a contiguous period of time in which the mobile is scheduled to receive data from the router). If the corresponding bit in the field mask is set, this parameter is considered valid only if the value is greater than zero.

This parameter is supported by WLAN networks only and ignored by all other network technologies.

Maximum service interval

This parameter defines the maximum interval, in microseconds, between the start of two service periods (a contiguous period of time in which the mobile is scheduled to receive data from the router). If the corresponding bit in the field mask is set, this parameter is considered valid only if the value is greater than zero. The application may use 0xFFFFFFFF to specify a value of infinity.

This parameter is supported by WLAN networks only and ignored by all other network technologies.

Inactivity interval

This parameter defines the minimum interval, in microseconds, which may elapse without arrival or transfer of data matching the filters specified in the QoS specification. Once this interval is elapsed, the router may release the QoS. If the corresponding bit in the field mask is set, this parameter is considered valid only if the value is greater than zero. The application may use 0xFFFFFFFF to specify a value of infinity.

This parameter is supported by WLAN networks only and ignored by all other network technologies.

IMS signalling context

This parameter indicates that the application wants to establish this QoS instance for IMS signalling. This parameter is supported by 3GPP networks only and ignored by all other network technologies.

High priority data

This parameter indicates that the data on this QoS instance has high priority and requests the network to expedite processing of the data. This parameter is supported by 3GPP networks only and ignored by all other network technologies.

Example 1

```
/*-----------------------------------------------------------------
In this example, application defines one flow for Rx direction and one
for Tx direction. Rx flow belongs to streaming class with required maximum
data rate of 64 kbps and guaranteed rate of 32 kbps. Tx flow is also of
type streaming class but is characterized by token bucket with a peak
rate of 128 kbps, token bucket rate of 32 kbps and token bucket size
of 2 maximum sized packets.
-----------------------------------------------------------------*/
ip_flow_type rx_flow;
ip_flow_type tx_flow;

  memset(&rx_flow, 0, sizeof(ip_flow_type));

  rx_flow.field_mask = IPFLOW_MASK_TRF_CLASS |
                       IPFLOW_MASK_DATA_RATE;

  rx_flow.trf_class = IP_TRF_CLASS_STREAMING;

  rx_flow.data_rate.format_type = DATA_RATE_FORMAT_MIN_MAX_TYPE;
  rx_flow.data_rate.format.min_max.max_rate = 64000;  /* 64 kbps */
  rx_flow.data_rate.format.min_max.guaranteed_rate = 32000;   /* 32 kbps */

  memset(&tx_flow, 0, sizeof(ip_flow_type));

  tx_flow.field_mask = IPFLOW_MASK_TRF_CLASS |
                       IPFLOW_MASK_DATA_RATE |
                       IPFLOW_MASK_LATENCY |
                       IPFLOW_MASK_MAX_ALLOWED_PKT_SIZE;

  tx_flow.trf_class = IP_TRF_CLASS_STREAMING;
  tx_flow.data_rate.format_type = DATA_RATE_FORMAT_TOKEN_BUCKET_TYPE;
  tx_flow.data_rate.format.token_bucket.peak_rate = 128000;  /* 128 kbps */
  tx_flow.data_rate.format.token_bucket.token_rate = 32000;  /* 32 kbps */
  tx_flow.data_rate.format.token_bucket.size = 3000;      /* 3000 bytes */
  tx_flow.latency = 500;  /* 500 msec */
  tx_flow. max_allowed_pkt_size = 1500;  /* 1500 bytes */
```

Example 2

```
/*-------------------------------------------------------------------
In this example, application defines one flow for Rx direction. Rx flow
belongs
to streaming class with specified 3GPP specific QoS parameters. If flow
shown
in this example is used on 3GPP2 system, 3GPP specific QoS parameters are
ignored.
-------------------------------------------------------------------*/
ip_flow_type rx_flow;

  memset(&rx_flow, 0, sizeof(ip_flow_type));

  rx_flow.field_mask = IPFLOW_MASK_TRF_CLASS |
                       IPFLOW_MASK_UMTS_RES_BER;

  rx_flow.trf_class = IP_TRF_CLASS_STREAMING;
  rx_flow.umts_params.res_ber = UMTS_RES_BIT_ERR_RATE1;
```

Example 3

```
/*-------------------------------------------------------------------
In this example, application defines one flow for Tx direction. Tx flow
belongs
to streaming class which uses 3GPP2 specific QoS parameters. If flow shown
in
this example is used on 3GPP system, 3GPP2 specific QoS parameters are
ignored.
-------------------------------------------------------------------*/
ip_flow_type tx_flow;

  memset(&tx_flow, 0, sizeof(ip_flow_type));

  tx_flow.field_mask = IPFLOW_MASK_TRF_CLASS |
                       IPFLOW_MASK_CDMA_PROFILE_ID;

  tx_flow.trf_class = IP_TRF_CLASS_STREAMING;
  tx_flow.cdma_params.profile_id = CDMA_QOS_PROFILE_32K_STREAMING;
```

6.3.3 Filter specification

This section describes the IP filter specification parameters and their semantics. The IP filter specification, when applied to a stream of IP packets, segregates a specific IP flow that is characterized by the filter specification parameters. The following structure defines the filter specification parameters supported by the DSS API.

Filter specification block definition

```
/*-------------------------------------------------------------------
Definition of filter specification block which contains filtering
parameters
required to segregate various IP flows.
-------------------------------------------------------------------*/
/*-------------------------------------------------------------------
   Mask of filtering parameters specified
-------------------------------------------------------------------*/

/* IPV4 hdr fields */
typedef enum
{
  IPFLTR_MASK_IP4_NONE,
  IPFLTR_MASK_IP4_SRC_ADDR,
  IPFLTR_MASK_IP4_DST_ADDR,
  IPFLTR_MASK_IP4_NEXT_HDR_PROT,
  IPFLTR_MASK_IP4_TOS,
  IPFLTR_MASK_IP4_ALL
} ipfltr_ip4_hdr_field_mask_enum_type;

/* IPV6 hdr fields */
typedef enum
{
  IPFLTR_MASK_IP6_NONE,
  IPFLTR_MASK_IP6_SRC_ADDR,
  IPFLTR_MASK_IP6_DST_ADDR,
  IPFLTR_MASK_IP6_NEXT_HDR_PROT,
  IPFLTR_MASK_IP6_TRAFFIC_CLASS,
  IPFLTR_MASK_IP6_FLOW_LABEL,
  IPFLTR_MASK_IP4_ALL
} ipfltr_ip6_hdr_field_mask_enum_type;

/* Higher level protocol hdr parameters */

/* TCP hdr fields */
typedef enum
```

```
{
  IPFLTR_MASK_TCP_NONE,
  IPFLTR_MASK_TCP_SRC_PORT,
  IPFLTR_MASK_TCP_DST_PORT,
  IPFLTR_MASK_TCP_ALL
} ipfltr_tcp_hdr_field_mask_enum_type;

/* UDP hdr fields */
typedef enum
{
  IPFLTR_MASK_UDP_NONE,
  IPFLTR_MASK_UDP_SRC_PORT,
  IPFLTR_MASK_UDP_DST_PORT,
  IPFLTR_MASK_UDP_ALL
} ipfltr_udp_hdr_field_mask_enum_type;

/* ICMP hdr fields */
typedef enum
{
  IPFLTR_MASK_ICMP_NONE,
  IPFLTR_MASK_ICMP_MSG_TYPE,
  IPFLTR_MASK_ICMP_MSG_CODE,
  IPFLTR_MASK_ICMP_ALL
} ipfltr_icmp_hdr_field_mask_enum_type;

typedef uint8 ipfltr_ip4_hdr_field_mask_type;
typedef uint8 ipfltr_ip6_hdr_field_mask_type;
typedef uint8 ipfltr_tcp_hdr_field_mask_type;
typedef uint8 ipfltr_udp_hdr_field_mask_type;
typedef uint8 ipfltr_icmp_hdr_field_mask_type;

typedef struct
{
  /* Mandatory Parameter - IP version of the filter (v4 or v6)    */
  ip_version_enum_type  ip_vsn;

  /* Filter parameter values, the ones set in field masks are only valid */
  /* Corresponding err mask is set if a parameter value is invalid */
  union
  {
    struct
    {
      ipfltr_ip4_hdr_field_mask_type      field_mask;  /* In mask  */
      ipfltr_ip4_hdr_field_mask_type      err_mask;    /* Out mask */
```

```
1
2               struct
3               {
4                 uint32  addr;
5                 uint32  subnet_mask;
6               } src;
7
8               struct
9               {
10                uint32  addr;
11                uint32  subnet_mask;
12              } dst;
13
14              struct
15              {
16                uint8 val;
17                uint8 mask;
18              } tos;
19
20              uint8 next_hdr_prot;
21            } v4;
22
23
24            struct
25            {
26              ipfltr_ip6_hdr_field_mask_type      field_mask;  /* In mask  */
27              ipfltr_ip6_hdr_field_mask_type      err_mask;    /* Out mask */
28
29              struct
30              {
31                uint64  addr[2];
32                uint8   prefix_len;
33              } src;
34
35              struct
36              {
37                uint64  addr[2];
38                uint8   prefix_len;
39              } dst;
40
41              struct
42              {
43                uint8   val;
44                uint8   mask;
```

```
      } trf_cls;

      uint32   flow_label;
      uint8    next_hdr_prot;
    } v6;
  } ip_hdr;

  /* next_hdr_prot field in v4 or v6 hdr must be set to specify a  */
  /* parameter from the next_prot_hdr                              */
  union
  {
    struct
    {
      ipfltr_tcp_hdr_field_mask_type    field_mask;  /* In mask  */
      ipfltr_tcp_hdr_field_mask_type    err_mask;    /* Out mask */

      struct
      {
        uint16    port;
        uint16    range;
      } src;

      struct
      {
        uint16    port;
        uint16    range;
      } dst;
    } tcp;


    struct
    {
      ipfltr_udp_hdr_field_mask_type     field_mask;  /* In mask  */
      ipfltr_udp_hdr_field_mask_type     err_mask;    /* Out mask */

      struct
      {
        uint16    port;
        uint16       range;
      } src;

      struct
      {
        uint16    port;
```

```
            uint16    range;
          } dst;
        } udp;

        struct
        {
          ipfltr_icmp_hdr_field_mask_type     field_mask; /* In mask   */
          ipfltr_icmp_hdr_field_mask_type     err_mask;   /* Out mask  */

          uint8   type;
          uint8   code;
        } icmp;

      } next_prot_hdr;
    } ip_filter_type;

    typedef struct
    {
      uint8             num_filters;      /* Num filters in the list   */
      ip_filter_type    *list_ptr;        /* List of filters           */
    } ip_filter_spec_type;
```

→	ip_filter_spec_type		
	→	num_filters	Number of filters specified in the filter specification block
	→	list_ptr	Pointer to an array containing one or more filters (as specified by num_filters); each filter is of type ip_filter_type as defined below
→	ip_filter_type		
	→	ip_vsn	IP version associated with the filter This is a mandatory parameter and each filter must be associated with an IP version (v4 or v6), and only the parameters defined by that IP version can be specified by the filter. IP version is of type ip_version_enum_type and the following values are defined: ▪ IP_V4 – Indicates an IPv4 filter ▪ IP_V6 – Indicates an IPv6 filter
	→	ip_hdr	Filter parameters that reside in IP header portion of an IP packet There is a separate parameter list for each IP version supported (currently v4 and v6).
	→	ip_hdr.v4	IP header parameters specific to IP version 4 These parameters are only valid if ip_vsn is set to IP_V4 or else they are ignored.

	→	`ip_hdr.v4.field_mask`	Bit mask to indicate which of the IPv4 parameters are specified If a bit is set, the value of the corresponding parameter is used from the filter structure. If a bit is not set, the corresponding parameter is ignored and is not applicable for filtering. The bit mask is of type ipfltr_ip4_hdr_field_mask_type and the following bits are defined: ▪ IPFLTR_MASK_IP4_SRC_ADDR – Corresponds to source IPv4 address ▪ IPFLTR_MASK_IP4_DST_ADDR – Corresponds to destination IPv4 address ▪ IPFLTR_MASK_IP4_NEXT_HDR_PROT – Corresponds to protocol field of the IPv4 header ▪ IPFLTR_MASK_IP4_SRC_TOS – Corresponds to type of service field of IPv4 header
	←	`ip_hdr.v4.err_mask`	This bit mask is returned to the caller to indicate errors in the specified IPv4 filter parameters. If a bit is set, the value of the corresponding parameter is erroneously specified, e.g., it could be out of the allowed range. A clear bit implies that either the corresponding flow parameter was not specified (i.e., the bit was not set in the field_mask, or the specified flow parameter value is correct. The bit mask is of type ipfltr_ip4_hdr_field_mask_type, which is described above. Callers should only pay attention to this field if an error is returned to the caller by the API in which the ip_filter_type structure is passed in as a parameter. If the API returned SUCCESS to the caller, this field is not valid and must be ignored.
	→	`ip_hdr.v4.src`	IPv4 source address – Specified by a combination of two fields, addr and subnet_mask Filtering is performed by comparing THE addr field logical ANDed with the subnet_mask against the source address from an IP packet logical ANDed with the subnet_mask. subnet_mask allows callers to set up a filter with a range of the source addresses, if needed. A subnet_mask of all 1s (255.255.255.255) specifies a single address value.
	→	`ip_hdr.v4.dst`	IPv4 destination address – Specified by a combination of two fields, addr and subnet_mask Filtering is performed by comparing the addr field logical ANDed with the subnet_mask against the destination address from an IP packet logical ANDed with the subnet_mask. subnet_mask allows callers to set up a filter with a range of the destination addresses, if needed. A subnet_mask of all 1s (255.255.255.255) specifies a single address value.

	→	`ip_hdr.v4.tos`	IPv4 type of service field – Specified by a combination of two fields, val and mask Filtering is performed by comparing val field logical ANDed with the mask against the TOS field from an IP packet logical ANDed with the mask. The mask should have a bit set if the corresponding bit in the TOS field in the IP packet should be considered for a match. The val should have a bit set if the corresponding bit in the TOS field in the IP packet should also be set and vice versa. In other words, only those bits in val are matched against the TOS field bits in the IP packet for which the corresponding bit is on in the mask. Example: ▪ val = 00101000 ▪ mask = 11111100 The filter will compare only the first 6 bits in the val with the first 6 bits in the TOS field of the IP packet. The first 6 bits in the TOS field of the IP packet must be 001010 to match the filter. The last 2 bits can be anything since they are ignored by filtering.
	→	`ip_hdr.v4.next_hdr_prot`	IPv4 protocol field – Identifies the higher layer protocol that needs to be considered for filtering an IP packet If this field is specified, only IP packets belonging to the specified higher layer protocol are considered for filtering. The filtering can be further enhanced by specifying parameters from that protocol header fields as described below. Only parameters from the next_hdr_prot will be considered, and other protocol header fields will be ignored. Protocol values are of the type ip_protocol_enum_type and THE following protocols are currently supported for QoS filtering: ▪ IPROTO_TCP – Protocol is TCP (IETF assigned value 6) ▪ IPROTO_UDP – Protocol is UDP (IETF assigned value 17) ▪ IPROTO_ICMP– Protocol is ICMP (IETF assigned value 1)
	→	`ip_hdr.v6`	IP header parameters specific to IP version 6 These parameters are only valid if ip_vsn is set to IP_V6 or else they are ignored.

	→	`ip_hdr.v6.field_mask`	Bit mask to indicate which of the IPv6 parameters are specified If a bit is set, the value of the corresponding parameter is used from the filter structure. If a bit is not set, the corresponding parameter is ignored and is not applicable for filtering. The bit mask is of type ipfltr_ip6_hdr_field_mask_type and the following bits are defined: ▪ IPFLTR_MASK_IP6_SRC_ADDR – Corresponds to source IPv6 address ▪ IPFLTR_MASK_IP6_DST_ADDR – Corresponds to destination IPv6 address ▪ IPFLTR_MASK_IP6_NEXT_HDR_PROT – Corresponds to next header field of the IPv6 header ▪ IPFLTR_MASK_IP6_TRAFFIC_CLASS – Corresponds to traffic class field of IPv6 header ▪ IPFLTR_MASK_IP6_FLOW_LABEL – Corresponds to flow label field of IPv6 header
	←	`ip_hdr.v6.err_mask`	Bit mask – Returned to the caller to indicate errors in the specified IPv6 filter parameters If a bit is set, the value of the corresponding parameter is erroneously specified, e.g., it could be out of the allowed range. A clear bit implies that either the corresponding flow parameter was not specified, i.e., the bit was not set in the field_mask or the specified flow parameter value is correct. The bit mask is of type ipfltr_ip6_hdr_field_mask_type, which is described above. Callers should only pay attention to this field if an error is returned to the caller by the API in which ip_filter_type structure is passed in as a parameter. If the API returned success to the caller, this field is not valid and must be ignored.
	→	`ip_hdr.v6.src`	IPv6 source address – Specified by a combination of two fields, addr and prefix_len Filtering is performed by comparing only the prefix_len number of leftmost bits from the addr field against the prefix_len number of leftmost bits from the source address in an IP packet. prefix_len allows callers to set up a filter with a range of the source addresses, if needed. A prefix_len of 128 specifies a single address value.
	→	`ip_hdr.v6.dst`	IPv6 destination address – Specified by a combination of two fields, addr and prefix_len Filtering is performed by comparing only the prefix_len number of leftmost bits from the addr field against the prefix_len number of leftmost bits from the destination address in an IP packet. prefix_len allows callers to set up a filter with a range of the destination addresses, if needed. A prefix_len of 128 specifies a single address value.

	→	ip_hdr.v6.trf_cls	IPv6 traffic class field – Specified by a combination of two fields, val and mask Filtering is performed by comparing val field logical ANDed with the mask against the traffic class field from an IP packet logical ANDed with the mask. The mask should have a bit set if the corresponding bit in the traffic class field in the IP packet should be considered for a match. The val should have a bit set if the corresponding bit in the traffic class field in the IP packet should also be set and vice versa. In other words, only those bits in val are matched against the traffic class field bits in the IP packet for which the corresponding bit is on in the mask. Example: ▪ val = 00101000 ▪ mask = 11111100 The filter will compare only the first 6 bits in the val with the first 6 bits in the traffic class field of the IP packet. The first 6 bits in the traffic class field of the IP packet must be 001010 to match the filter. The last 2 bits can be anything since they are ignored by filtering.
	→	ip_hdr.v6.flow_label	IPv6 flow label field
	→	ip_hdr.v6.next_hdr_prot	IPv6 next header field – Identifies the higher layer protocol that needs to be considered for filtering an IP packet If this field is specified, only IP packets belonging to the specified higher layer protocol are considered for filtering. The filtering can be further enhanced by specifying parameters from that protocol header fields as described below. Only parameters from the next_hdr_prot will be considered and other protocol header fields will be ignored. Protocol values are of the type ip_protocol_enum_type and the following protocols are currently supported for QoS filtering: ▪ IPROTO_TCP – Protocol is TCP (IETF assigned value 6) ▪ IPROTO_UDP – Protocol is UDP (IETF assigned value 17) ▪ IPROTO_ICMP – Protocol is ICMP (IETF assigned value 1)
	→	next_prot_hdr	Filter parameters that reside in a higher-layer header portion of an IP packet There is a separate parameter list for each higher-layer header. Parameters corresponding to only one higher layer header can be specified, and the header corresponds to the protocol specified by ip_hdr.v4.next_hdr_prot for IPv4 packets and ip_hdr.v6.next_hdr_prot IPv6 packets.
	→	next_prot_hdr.tcp	Filter parameters specific to TCP header portion of the IP packet These parameters are only valid if next_hdr_prot in ip_hdr is set to IPPROTO_TCP.

	→	`next_prot_hdr.tcp.field_mask`	Bit mask to indicate which of the TCP parameters are specified If a bit is set, the value of the corresponding parameter is used from the filter structure. If a bit is not set, the corresponding parameter is ignored and is not applicable for filtering. The bit mask is of type ipfltr_tcp_hdr_field_mask_type and the following bits are defined: ▪ IPFLTR_MASK_TCP_SRC_PORT – Corresponds to TCP source port ▪ IPFLTR_MASK_TCP_DST_PORT – Corresponds to TCP destination port
	←	`next_prot_hdr.tcp.err_mask`	Bit mask returned to the caller to indicate errors in the specified TCP filter parameters If a bit is set, the value of the corresponding parameter is erroneously specified, e.g., it could be out of allowed range. A clear bit implies that either the corresponding flow parameter was not specified, i.e., the bit was not set in the field_mask or the specified flow parameter value is correct. The bit mask is of type ipfltr_tcp_hdr_field_mask_type, which is described above. Callers should only pay attention to this field if an error is returned to the caller by the API in which the ip_filter_type structure is passed in as a parameter. If the API returned SUCCESS to the caller, this field is not valid and must be ignored.
	→	`next_prot_hdr.tcp.src`	TCP source port – Specified by a combination of two fields, port and range The value of range specifies the number of ports to be included in the filter starting from the value port. The filter will match if the TCP source port in the IP packet lies between port and port+range. A range value of 0 implies that only one value of the TCP source port is valid as specified by the port.
	→	`next_prot_hdr.tcp.dst`	TCP destination port – Specified by a combination of two fields, port and range The value of range specifies the number of ports to be included in the filter starting from the value port. The filter will match if the TCP destination port in the IP packet lies between port and port+range. A range value of 0 implies that only one value of the TCP destination port is valid as specified by the port.
	→	`next_prot_hdr.udp`	Filter parameters specific to UDP header portion of the IP packet These parameters are only valid if next_hdr_prot in ip_hdr is set to IPPROTO_UDP.
	→	`next_prot_hdr.udp.field_mask`	Bit mask to indicate which of the UDP parameters are specified If a bit is set, the value of the corresponding parameter is used from the filter structure. If a bit is not set, the corresponding parameter is ignored and is not applicable for filtering. The bit mask is of type ipfltr_udp_hdr_field_mask_type and the following bits are defined: ▪ IPFLTR_MASK_UDP_SRC_PORT – Corresponds to UDP source port ▪ IPFLTR_MASK_UDP_DST_PORT – Corresponds to UDP destination port

	←	next_prot_hdr.udp.err_mask	Bit mask returned to the caller to indicate errors in the specified UDP filter parameters If a bit is set, the value of the corresponding parameter is erroneously specified, e.g., it could be out of allowed range. A clear bit implies that either the corresponding flow parameter was not specified, i.e., the bit was not set in the field_mask or the specified flow parameter value is correct. The bit mask is of type ipfltr_udp_hdr_field_mask_type, which is described above. Callers should only pay attention to this field if an error is returned to the caller by the API in which the ip_filter_type structure is passed in as a parameter. If the API returned SUCCESS to the caller, this field is not valid and must be ignored.
	→	next_prot_hdr.udp.src	UDP source port – Specified by a combination of two fields, port and range The value of range specifies the number of ports to be included in the filter starting from the value port. The filter will match if the UDP source port in the IP packet lies between port and port+range. A range value of 0 implies that only one value of the UDP source port is valid as specified by the port.
	→	next_prot_hdr.udp.dst	UDP destination port – Specified by a combination of two fields, port and range The value of range specifies the number of ports to be included in the filter starting from the value port. The filter will match if the UDP destination port in the IP packet lies between port and port+range. A range value of 0 implies that only one value of the UDP destination port is valid as specified by port.
	→	next_prot_hdr.icmp	Filter parameters specific to ICMP header portion of the IP packet These parameters are only valid if next_hdr_prot in ip_hdr is set to IPPROTO_ICMP.
	→	next_prot_hdr.icmp.field_mask	Bit mask to indicate which of the ICMP parameters are specified If a bit is set, the value of the corresponding parameter is used from the filter structure. If a bit is not set, the corresponding parameter is ignored and is not applicable for filtering. The bit mask is of type ipfltr_icmp_hdr_field_mask_type and the following bits are defined: ▪ IPFLTR_MASK_ICMP_MSG_TYPE – Corresponds to ICMP message type field ▪ IPFLTR_MASK_ICMP_MSG_CODE – Corresponds to ICMP message code field

	←	next_prot_hdr.icmp.err_mask	Bit mask returned to the caller to indicate errors in the specified ICMP filter parameters If a bit is set, the value of the corresponding parameter is erroneously specified, e.g., it could be out of the allowed range. A clear bit implies that either the corresponding flow parameter was not specified, i.e., the bit was not set in the field_mask or the specified flow parameter value is correct. The bit mask is of type ipfltr_icmp_hdr_field_mask_type, which is described above. Callers should only pay attention to this field if an error is returned to the caller by the API in which the ip_filter_type structure is passed in as a parameter. If the API returned success to the caller, this field is not valid and must be ignored.
	→	next_prot_hdr.icmp.type	ICMP message type field See [**Error! Reference source not found.**] for the allowed values for this field.
	→	next_prot_hdr.icmp.code	ICMP message code field See [**Error! Reference source not found.**] for the allowed values for this field.

NOTE: IPv6 is not supported and QoS is not guaranteed if v6 filters are installed. v6 fields are defined so that structures need not change when support for QoS is added.

NOTE: ICMP is not supported and filters are rejected if they contain ICMP-specific parameters.

6.3.3.1 Filter specification semantics

This section describes the validation rules and semantics for the IP filter specification parameters.

1. A bit in field_mask shall be set for each parameter specified; if a bit is not set, the corresponding parameter is ignored.
2. If one or more parameters are incorrrectly specified, the bits in err_mask indicate those parameters on return. Hence, in case of errors, err_mask should be checked to see if a parameter validation error has occurred that will be otherwise cleared.
3. If a parameter consisting of multiple subfields is specified, each subfield shall also be specified. Such parameters are:
   - IP v4/v6 src and dst addr
   - IP v4 TOS
   - IP v6 traffic class
   - TCP/UDP src and dst port
4. Each filter will have an IP version associated with it, either v4 or v6, and shall be specified by ip_vsn. It is invalid to specify v6 parameters for a v4 type filter and vice versa.
5. Filters belonging to either IPv4 or IPv6 may be installed on an iface, but only the filters belonging to the current iface IP family (v4 or v6) are used/executed and remaining filters are ignored.
6. Not all combinations of filter parameters are allowed in each filter. Table 6-1 describes the valid combinations. Only those attributes marked with an X may be specified for a single packet filter. All marked attributes may be specified, but at least one shall be specified.

Table 6-1 Valid packet filter attribute combinations

Packet filter attribute	Valid combination types			
	I	II	III	IV
Source Address and Subnet Mask	X	X	X	X
Destination Address and Subnet Mask	X	X	X	X
Protocol Number (IPv4)/Next Header (IPv6)	X	X	X	
Destination Port Range	X			
Source Port Range	X			
TOS (IPv4)/Traffic Class (IPv6) and Mask	X	X	X	X
Flow Label (IPv6)				X

7. If a parameter from the next header protocol (TCP, UDP, etc.) is specified in a filter, the protocol number for v4 filters or next header for v6 filters shall be specified. The only protocols currently supported are TCP and UDP.
8. In most cases the IP source address in Tx filters and IP destination address in Rx filters is not required to be specified. In the case where these values are specified, the following requirements shall be met:
   - The IP source address in Tx filters and the IP destination address in Rx filters can only have a single address value. Hence, for v4 filters, subnet_mask must be 0xFFFFFFFF, and for v6 filters, prefix_len must be 128.
   - These address values shall be same as the current IP address assigned to the MS on the IP interface on which the QoS is being requested.
   - If the IP address on the interface changes e.g., during a network handoff, the client that requested the QoS is responsible for updating the filters with the new address, or else the flow may not be able to receive the desired treatment.
9. The IP destination address in Tx filters and the IP source address in Rx filters can be specified as address ranges using the subnet_mask for v4 filters or prefix_len for v6 filters parameters. A subnet_mask may be set to all 1s or prefix_len may be set to 128 to specify a single address value.
10. Only a nonnegative and nonzero value can be used as IPv4 tos or IPv6 trf_cls.
11. Only a nonnegative and nonzero value can be used as a port.
12. A nonnegative and nonzero value for range may be set to specify a range of port numbers starting from the value specified in port [port to port+range], otherwise range shall be set to 0. (port+range) must be less than 65535 (216 - 1).
13. There may be additional validations that each network technology handler may impose. For details on network-specific semantics, refer to the respective standards documentation.
14. Certain fields like address, port numbers, etc., shall be specified in network byte order; everything else shall be in host byte order. The following fields shall be specified in netwok byte order:
    - IPv4 addresses

- IPv4 subnet masks
- IPv6 addresses (prefix length shall be in host order)
- TCP and UDP port numbers (port ranges shall be in host order)
- IPv6 flow label

Example 4

```
/*-----------------------------------------------------------------
In this example, application defines one filter for Rx direction and one
for Tx direction. Rx filter captures IPv4 UDP packets sent from
MY_SERVER_PORT
port. Tx filter captures IPv4 TCP packets sent to a host with IP
address MY_SERVER_ADDR and port MY_SERVER_PORT.
-----------------------------------------------------------------*/
#define MY_SERVER_ADDR    0xC0A80E02    /* 192.168.14.2 */
#define MY_SERVER_PORT    5000

ip_filter_type rx_fltr;
ip_filter_type tx_fltr;

  memset(&rx_fltr, 0, sizeof(ip_filter_type));
  rx_fltr.ip_vsn = IP_V4;

  rx_fltr.ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT;
  rx_fltr.ip_hdr.v4.next_hdr_prot = IPPROTO_UDP;

  rx_fltr.next_prot_hdr.udp.field_mask = IPFLTR_MASK_UDP_SRC_PORT;
  rx_fltr.next_prot_hdr.udp.src.port = dss_htons(MY_SERVER_PORT);
  rx_fltr.next_prot_hdr.udp.src.range = 0;

  memset(&tx_fltr, 0, sizeof(ip_filter_type));
  tx_fltr.ip_vsn = IP_V4;

  tx_fltr.ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT |
                                 IPFLTR_MASK_IP4_DST_ADDR;
  tx_fltr.ip_hdr.v4.next_hdr_prot = IPPROTO_TCP;
  tx_fltr.ip_hdr.v4.dst.addr = dss_htonl(MY_SERVER_ADDR);
  tx_fltr.ip_hdr.v4.dst.subnet_mask = dss_htonl(0xFFFFFFFF);

  tx_fltr.next_prot_hdr.tcp.field_mask = IPFLTR_MASK_TCP_DST_PORT;
  tx_fltr.next_prot_hdr.tcp.dst.port = dss_htons(MY_SERVER_PORT);
  tx_fltr.next_prot_hdr.tcp.dst.port = 0;
```

Example 5

```
/*---------------------------------------------------------------------
In this example, application defines one filter for Rx direction and one
for Tx direction. Rx filter captures IPv6 UDP packets sent from a range of
ports
starting from MY_SERVER_PORT. Tx filter captures IPv4 packets sent to a
host
with IP address MY_SERVER_ADDR.
---------------------------------------------------------------------*/
#define MY_SERVER_ADDR          0xC0A80E02    /* 192.168.14.2 */
#define MY_SERVER_PORT          5000
#define MY_SERVER_PORT_RANGE    10

ip_filter_type rx_fltr;
ip_filter_type tx_fltr;

  memset(&rx_fltr, 0, sizeof(ip_filter_type));
  rx_fltr.ip_vsn = IP_V6;

  rx_fltr.ip_hdr.v6.field_mask = IPFLTR_MASK_IP6_NEXT_HDR_PROT;
  rx_fltr.ip_hdr.v6.next_hdr_prot = IPPROTO_UDP;

  rx_fltr.next_prot_hdr.udp.field_mask = IPFLTR_MASK_UDP_SRC_PORT;
  rx_fltr.next_prot_hdr.udp.src.port = dss_htons(MY_SERVER_PORT);
  rx_fltr.next_prot_hdr.udp.src.range = MY_SERVER_PORT_RANGE;

  memset(&tx_fltr, 0, sizeof(ip_filter_type));
  tx_fltr.ip_vsn = IP_V4;

  tx_fltr.ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_DST_ADDR;
  tx_fltr.ip_hdr.v4.dst.addr = dss_htonl(MY_SERVER_ADDR);
  tx_fltr.ip_hdr.v4.dst.subnet_mask = dss_htonl(0xFFFFFFFF);
```

Example 6

```
/*---------------------------------------------------------------------
In this example, application defines one filter for Rx direction. Rx filter
captures all IPv4 UDP packets sent to mobile
---------------------------------------------------------------------*/

ip_filter_type rx_fltr;
```

```
memset(&rx_fltr, 0, sizeof(ip_filter_type));
rx_fltr.ip_vsn = IP_V4;

rx_fltr.ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT;

rx_fltr.ip_hdr.v4.next_hdr_prot = IPPROTO_UDP;
```

6.3.3.2 Ordering of filters

The order in which filters are installed on the network is very important in deciding the kind of treatment a packet receives on the Rx path. The network does not know which application a packet is destined for and, hence, it runs a packet through all the installed filters in a prespecified order, called precedence. Each filter is assigned a unique precedence by the mobile device before installing filters on the network. The precedence is unique among all the filters for all the active QoS requests on a given network interface. Filters are executed in the order of precedence, which influences the flow to which a packet will be assigned. AMSS QoS implementation assigns filter precedence based on an FCFS basis. Filters associated with the first QoS request get the higher precedence as compared to the filters associated with a subsequent QoS request. All filters associated with a given QoS request are assigned a contiguous precedence value.

6.3.3.3 Semantics of filter matching for Network Initiated QoS

Network Initiated QoS shall be considered “matching” to application request in DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST if the filters specification provided by the application is identical or is a subset of the filters specification granted by the network. For example, if application specifies two filters while the network specifies those same two filters but additional filters (in the same direction) then that filter specification shall match the application request. Another example is if the application specifies a port number in one of the filters specification and the network filters specification indicates a port range that includes the port number specified by the application. This constitutes a match.

6.3.4 Sample QoS specifications

The following examples illustrate typical definitions of QoS specification.

Example 7

```
/*-----------------------------------------------------------------
In this example, application defines one flow in Rx direction and two flows
in Tx direction. Rx required flow belongs to streaming class. Tx required
flow
is also of type streaming class but is characterized by a latency of 500ms
and
maximum allowed packet size of 1500 bytes. Tx minimum required flow is
characterized by latency variation of 50ms. Application also defines one
filter
for Rx direction and one for Tx direction. Rx filter captures IPv4 UDP
packets
```

```
1         sent from MY_SERVER_PORT port. Tx filter captures IPv4 TCP packets sent to
2         a
3         host with IP address MY_SERVER_ADDR and port MY_SERVER_PORT. This QoS
4         specification is used in DSS_IFACE_IOCTL_QOS_REQUEST operation.
5         ---------------------------------------------------------------------*/
6
7         #define MY_SERVER_ADDR    0xC0A80E02    /* 192.168.14.2 */
8         #define MY_SERVER_PORT    5000
9         #define NUM_RX_FLTRS      1
10        #define NUM_TX_FLTRS      1
11
12        int qos_request()
13        {
14        qos_spec_type                         qos_spec;
15        ip_flow_type                        * ip_flow_ptr;
16        ip_filter_type                        rx_fltr[NUM_RX_FLTRS];
17        ip_filter_type                        tx_fltr[NUM_TX_FLTRS];
18
19          memset(&qos_spec, 0, sizeof(qos_spec_type));
20
21          qos_spec.field_mask = QOS_MASK_RX_FLOW | QOS_MASK_TX_FLOW |
22                                QOS_MASK_TX_MIN_FLOW;
23          /*-----------------------------------------------------------------
24            Populate Rx flow template
25          -----------------------------------------------------------------*/
26          ip_flow_ptr = &(qos_spec.rx.flow_template.req_flow);
27          ip_flow_ptr->field_mask = IPFLOW_MASK_TRF_CLASS;
28
29          ip_flow_ptr->trf_class = IP_TRF_CLASS_STREAMING;
30
31          /*-----------------------------------------------------------------
32            Populate Rx fltr template
33          -----------------------------------------------------------------*/
34          memset(&rx_fltr, 0, sizeof(ip_filter_type) * NUM_RX_FLTRS);
35
36          rx_fltr[0].ip_vsn = IP_V4;
37
38          rx_fltr[0].ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT;
39          rx_fltr[0].ip_hdr.v4.next_hdr_prot = IPPROTO_UDP;
40
41          rx_fltr[0].next_prot_hdr.udp.field_mask = IPFLTR_MASK_UDP_SRC_PORT;
42          rx_fltr[0].next_prot_hdr.udp.src.port = dss_htons(MY_SERVER_PORT);
43          rx_fltr[0].next_prot_hdr.udp.src.range = 0;
44
```

```
        qos_spec.rx.fltr_template.num_filters = NUM_RX_FLTRS;
        qos_spec.rx.fltr_template.list_ptr = rx_fltr;

        /*-------------------------------------------------------------------
          Populate Tx flow template
        -------------------------------------------------------------------*/
        ip_flow_ptr = &(qos_spec.tx.flow_template.req_flow);

        ip_flow_ptr->field_mask = IPFLOW_MASK_TRF_CLASS |
                                  IPFLOW_MASK_LATENCY |
                                  IPFLOW_MASK_MAX_ALLOWED_PKT_SIZE;

        ip_flow_ptr->trf_class = IP_TRF_CLASS_STREAMING;
        ip_flow_ptr->latency = 500;  /* 500 msec */
        ip_flow_ptr->max_allowed_pkt_size = 1500;  /* 1500 bytes */

        ip_flow_ptr = &(qos_spec.tx.flow_template.min_req_flow);

        ip_flow_ptr->field_mask = IPFLOW_MASK_LATENCY_VAR;
        ip_flow_ptr->latency_var = 20;  /* 20 msec */


      /*-------------------------------------------------------------------
          Populate Tx fltr template
        -------------------------------------------------------------------*/
        memset(&tx_fltr, 0, sizeof(ip_filter_type) * NUM_TX_FLTRS);

        tx_fltr[0].ip_vsn = IP_V4;

        tx_fltr[0].ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT |
                                          IPFLTR_MASK_IP4_DST_ADDR;
        tx_fltr[0].ip_hdr.v4.next_hdr_prot = IPPROTO_TCP;
        tx_fltr[0].ip_hdr.v4.dst.addr = dss_htonl(MY_SERVER_ADDR);
        tx_fltr[0].ip_hdr.v4.dst.subnet_mask = dss_htonl(0xFFFFFFFF);

        tx_fltr[0].next_prot_hdr.tcp.field_mask = IPFLTR_MASK_TCP_DST_PORT;
        tx_fltr[0].next_prot_hdr.tcp.dst.port = dss_htons(MY_SERVER_PORT);
        tx_fltr[0].next_prot_hdr.tcp.dst.port = 0;

        qos_spec.tx.fltr_template.num_filters = NUM_TX_FLTRS;
        qos_spec.tx.fltr_template.list_ptr = tx_fltr;
```

Example 8

```
/*---------------------------------------------------------------------
In this example, application deletes QoS in Rx direction and defines two
flows
in Tx direction. Tx required flow is of type streaming class. Tx auxiliary
flow is of type interactive class. Application also defines one filter
in Tx direction. Tx filter captures IPv4 TCP packets sent to a host with IP
address MY_SERVER_ADDR and port MY_SERVER_PORT. This QoS specification is
used
with DSS_IFACE_IOCTL_QOS_MODIFY operation.
---------------------------------------------------------------------*/

#define MY_SERVER_ADDR    0xC0A80E02    /* 192.168.14.2 */
#define MY_SERVER_PORT    5000
#define NUM_TX_AUX_FLOWS  1
#define NUM_TX_FLTRS      1

int qos_request()
{
qos_spec_type                    qos_spec;
ip_flow_type                   * ip_flow_ptr;
ip_flow_type                     tx_aux_flow[NUM_TX_AUX_FLOWS];
ip_filter_type                   tx_fltr[NUM_TX_FLTRS];

  memset(&qos_spec, 0, sizeof(qos_spec_type));

  qos_spec.field_mask = QOS_MASK_RX_FLOW | QOS_MASK_TX_FLOW |
                        QOS_MASK_TX_AUXILIARY_FLOWS;

  /*-------------------------------------------------------------------
    Delete Rx flow template
  -------------------------------------------------------------------*/
  ip_flow_ptr = &(qos_spec.rx.flow_template.req_flow);
  ip_flow_ptr->field_mask = IPFLOW_MASK_NONE;

  /*-------------------------------------------------------------------
    Populate Tx flow template
  -------------------------------------------------------------------*/
  ip_flow_ptr = &(qos_spec.tx.flow_template.req_flow);

  ip_flow_ptr->field_mask = IPFLOW_MASK_TRF_CLASS;
  ip_flow_ptr->trf_class = IP_TRF_CLASS_STREAMING;
  tx_aux_flow[0].field_mask = IPFLOW_MASK_TRF_CLASS;
  tx_aux_flow[0].trf_class = IP_TRF_CLASS_INTERACTIVE;
```

```
  qos_spec.tx.flow_template.num_aux_flows = NUM_TX_AUX_FLOWS;
  qos_spec.tx.flow_template.aux_flow_list_ptr = tx_aux_flow;

  /*-------------------------------------------------------------------
    Populate Tx fltr template
  -------------------------------------------------------------------*/
  memset(&tx_fltr, 0, sizeof(ip_filter_type) * NUM_TX_FLTRS);

  tx_fltr[0].ip_vsn = IP_V4;

  tx_fltr[0].ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT |
                                    IPFLTR_MASK_IP4_DST_ADDR;
  tx_fltr[0].ip_hdr.v4.next_hdr_prot = IPPROTO_TCP;
  tx_fltr[0].ip_hdr.v4.dst.addr = dss_htonl(MY_SERVER_ADDR);
  tx_fltr[0].ip_hdr.v4.dst.subnet_mask = dss_htonl(0xFFFFFFFF);

  tx_fltr[0].next_prot_hdr.tcp.field_mask = IPFLTR_MASK_TCP_DST_PORT;
  tx_fltr[0].next_prot_hdr.tcp.dst.port = dss_htons(MY_SERVER_PORT);
  tx_fltr[0].next_prot_hdr.tcp.dst.port = 0;

  qos_spec.tx.fltr_template.num_filters = NUM_TX_FLTRS;
  qos_spec.tx.fltr_template.list_ptr = tx_fltr;
```

Example 9

```
/*-------------------------------------------------------------------
In this example, application adds two flows in Rx direction. Rx minimum
required flow is characterized by a latency of 300ms. Rx auxiliary flow is
characterized by a latency of 300ms. Application also adds one more filter
in
Tx direction. Both Tx filters capture IPv4 TCP packets sent to a host with
IP
address MY_SERVER_ADDR and port MY_SERVER_PORT. This QoS specification is
used
with DSS_IFACE_IOCTL_QOS_MODIFY operation. Please also note that QoS is
previously requested in both Rx and Tx directions and that one filter is
installed in Tx direction.
-------------------------------------------------------------------*/

#define MY_SERVER_ADDR    0xC0A80E02    /* 192.168.14.2 */
#define MY_SERVER_PORT    5000
#define NUM_RX_AUX_FLOWS  1
#define NUM_TX_FLTRS      2
```

```
1
2         int qos_request()
3         {
4         qos_spec_type                          qos_spec;
5         ip_flow_type                         * ip_flow_ptr;
6         ip_flow_type                           rx_aux_flow[NUM_RX_AUX_FLOWS];
7         ip_filter_type                         tx_fltr[NUM_TX_FLTRS];
8
9           memset(&qos_spec, 0, sizeof(qos_spec_type));
10
11          qos_spec.field_mask = QOS_MASK_RX_FLOW | QOS_MASK_RX_MIN_FLOW |
12                                QOS_MASK_RX_AUXILIARY_FLOWS |
13                                QOS_MODIFY_MASK_TX_FLTR_MODIFY;
14
15          /*-----------------------------------------------------------------
16            Populate Rx flow template
17          -----------------------------------------------------------------*/
18          ip_flow_ptr = &(qos_spec.rx.flow_template.req_flow);
19          ip_flow_ptr->field_mask = IPFLOW_MASK_LATENCY;
20          ip_flow_ptr->latency = 100;
21
22
23          ip_flow_ptr = &(qos_spec.rx.flow_template.min_req_flow);
24          ip_flow_ptr->field_mask = IPFLOW_MASK_LATENCY;
25          ip_flow_ptr->latency = 300;
26
27          rx_aux_flow[0].field_mask = IPFLOW_MASK_LATENCY;
28          rx_aux_flow[0].latency = 200;
29
30          qos_spec.rx.flow_template.num_aux_flows = NUM_RX_AUX_FLOWS;
31          qos_spec.rx.flow_template.aux_flow_list_ptr = rx_aux_flow;
32
33          /*-----------------------------------------------------------------
34            Populate Tx fltr template
35          -----------------------------------------------------------------*/
36          memset(&tx_fltr, 0, sizeof(ip_filter_type) * NUM_TX_FLTRS);
37
38          tx_fltr[0].ip_vsn = IP_V4;
39
40          tx_fltr[0].ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT |
41                                            IPFLTR_MASK_IP4_DST_ADDR;
42          tx_fltr[0].ip_hdr.v4.next_hdr_prot = IPPROTO_TCP;
43          tx_fltr[0].ip_hdr.v4.dst.addr = dss_htonl(MY_SERVER_ADDR);
44          tx_fltr[0].ip_hdr.v4.dst.subnet_mask = dss_htonl(0xFFFFFFFF);
```

```
  tx_fltr[0].next_prot_hdr.tcp.field_mask = IPFLTR_MASK_TCP_DST_PORT;
  tx_fltr[0].next_prot_hdr.tcp.dst.port = dss_htons(MY_SERVER_PORT);
  tx_fltr[0].next_prot_hdr.tcp.dst.port = 0;

  tx_fltr[1] = tx_fltr[0];

  qos_spec.tx.fltr_template.num_filters = NUM_TX_FLTRS;
  qos_spec.tx.fltr_template.list_ptr = tx_fltr;
```

Example 11

```
/*-----------------------------------------------------------------
In this example, application removes one filter in Tx direction. Tx filter
captures IPv4 TCP packets sent to a host with IP address MY_SERVER_ADDR and
port MY_SERVER_PORT. There is no change to either flow specification or
filter specification in Rx direction. This QoS specification is used with
DSS_IFACE_IOCTL_QOS_MODIFY operation. Please also note that QoS is
previously requested in both Rx and Tx directions and that two filters are
installed in Tx direction.
-----------------------------------------------------------------*/

#define MY_SERVER_ADDR    0xC0A80E02    /* 192.168.14.2 */
#define MY_SERVER_PORT    5000
#define NUM_TX_FLTRS      1

int qos_modify()
{
qos_spec_type                          qos_spec;
ip_flow_type                         * ip_flow_ptr;
ip_flow_type                           rx_aux_flow[NUM_RX_AUX_FLOWS];
ip_filter_type                         tx_fltr[NUM_TX_FLTRS];

  memset(&qos_spec, 0, sizeof(qos_spec_type));

  qos_spec.field_mask = QOS_MODIFY_MASK_TX_FLTR_MODIFY;

  /*-----------------------------------------------------------------
    Populate Tx fltr template
  -----------------------------------------------------------------*/
  memset(&tx_fltr, 0, sizeof(ip_filter_type) * NUM_TX_FLTRS);

  tx_fltr[0].ip_vsn = IP_V4;

  tx_fltr[0].ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT |
```

```
                                        IPFLTR_MASK_IP4_DST_ADDR;
  tx_fltr[0].ip_hdr.v4.next_hdr_prot = IPPROTO_TCP;
  tx_fltr[0].ip_hdr.v4.dst.addr = dss_htonl(MY_SERVER_ADDR);
  tx_fltr[0].ip_hdr.v4.dst.subnet_mask = dss_htonl(0xFFFFFFFF);

  tx_fltr[0].next_prot_hdr.tcp.field_mask = IPFLTR_MASK_TCP_DST_PORT;
  tx_fltr[0].next_prot_hdr.tcp.dst.port = dss_htons(MY_SERVER_PORT);
  tx_fltr[0].next_prot_hdr.tcp.dst.port = 0;

  qos_spec.tx.fltr_template.num_filters = NUM_TX_FLTRS;
  qos_spec.tx.fltr_template.list_ptr = tx_fltr;
```

Example 12

```
/*-----------------------------------------------------------------
In this example, application removes two flows in Rx direction. Rx
required flow is characterized by a latency of 300ms. This QoS
specification is used with DSS_IFACE_IOCTL_QOS_MODIFY operation. Please
also note that QoS is previously requested in Rx direction only and that
required, minimum required and auxiliary flows are requested.
-----------------------------------------------------------------*/

int qos_modify()
{
qos_spec_type                       qos_spec;
ip_flow_type                      * ip_flow_ptr;

  memset(&qos_spec, 0, sizeof(qos_spec_type));

  qos_spec.field_mask = QOS_MASK_RX_FLOW;

  /*-----------------------------------------------------------------
    Populate Rx flow template
  -----------------------------------------------------------------*/
  ip_flow_ptr = &(qos_spec.rx.flow_template.req_flow);
  ip_flow_ptr->field_mask = IPFLOW_MASK_LATENCY;
  ip_flow_ptr->latency = 300;
```

6.3.5 Sample usage of QoS API

A typical usage of the QoS API is shown in the following example.

Example 13

```
/*-----------------------------------------------------------------
In this example, application defines one flow for Rx direction and one for
Tx direction. Rx flow belongs to streaming class with required maximum data
rate of 64 kbps and guaranteed rate of 32 kbps. Tx flow is also of type
streaming class but is characterized by a latency of 500ms and maximum
allowed packet size of 1500 bytes. Application also defines one filter for
Rx direction and one for Tx direction. Rx filter captures IPv4 UDP packets
sent from
MY_SERVER_PORT port. Tx filter captures IPv4 TCP packets sent to a
host with IP address MY_SERVER_ADDR and port MY_SERVER_PORT. After a while,
application modifies flow in Rx direction to request a maximum data rate of
128 kbps and guaranteed rate of 64 kbps.
-----------------------------------------------------------------*/
#define MY_SERVER_ADDR    0xC0A80E02    /* 192.168.14.2 */
#define MY_SERVER_PORT    5000
#define NUM_RX_FLTRS      1
#define NUM_TX_FLTRS      1

dss_iface_ioctl_event_enum_type qos_event;

int qos_request()
{
dss_iface_ioctl_qos_request_type       qos_request;
dss_iface_ioctl_qos_modify_type        qos_modify;
dss_iface_ioctl_qos_release_type       qos_release;
dss_iface_ioctl_qos_get_flow_spec_type qos_flow_spec;
qos_spec_type                          qos_spec;
qos_spec_type                          qos_modify_spec;
ip_flow_type                         * ip_flow_ptr;
ip_filter_type                         rx_fltr[NUM_RX_FLTRS];
ip_filter_type                         tx_fltr[NUM_TX_FLTRS];

  memset(&qos_spec, 0, sizeof(qos_spec_type));

  qos_spec.field_mask = QOS_MASK_RX_FLOW | QOS_MASK_TX_FLOW;

  /*-----------------------------------------------------------------
    Populate Rx flow template
  -----------------------------------------------------------------*/
  ip_flow_ptr = &(qos_spec.rx.flow_template.req_flow);
  ip_flow_ptr->field_mask = IPFLOW_MASK_TRF_CLASS |
```

```
                                IPFLOW_MASK_DATA_RATE;

      ip_flow_ptr->trf_class = IP_TRF_CLASS_STREAMING;

      ip_flow_ptr->data_rate.format_type = DATA_RATE_FORMAT_MIN_MAX_TYPE;
      ip_flow_ptr->data_rate.format.min_max.max_rate = 64000;
       /* 64 kbps */
      ip_flow_ptr-> data_rate.format.min_max.guaranteed_rate = 32000;
       /* 32 kbps */

      /*-----------------------------------------------------------------
        Populate Rx fltr template
      -----------------------------------------------------------------*/
      memset(&rx_fltr, 0, sizeof(ip_filter_type) * NUM_RX_FLTRS);

      rx_fltr[0].ip_vsn = IP_V4;

      rx_fltr[0].ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT;
      rx_fltr[0].ip_hdr.v4.next_hdr_prot = IPPROTO_UDP;

      rx_fltr[0].next_prot_hdr.udp.field_mask = IPFLTR_MASK_UDP_SRC_PORT;
      rx_fltr[0].next_prot_hdr.udp.src.port = dss_htons(MY_SERVER_PORT);
      rx_fltr[0].next_prot_hdr.udp.src.range = 0;

      qos_spec.rx.fltr_template.num_filters = NUM_RX_FLTRS;
      qos_spec.rx.fltr_template.list_ptr = rx_fltr;

      /*-----------------------------------------------------------------
        Populate Tx flow template
      -----------------------------------------------------------------*/
      ip_flow_ptr = &(qos_spec.tx.flow_template.req_flow);

      ip_flow_ptr->field_mask = IPFLOW_MASK_TRF_CLASS |
                                IPFLOW_MASK_LATENCY |
                                IPFLOW_MASK_MAX_ALLOWED_PKT_SIZE;

      ip_flow_ptr->trf_class = IP_TRF_CLASS_STREAMING;
      ip_flow_ptr->latency = 500;  /* 500 msec */
      ip_flow_ptr->max_allowed_pkt_size = 1500;  /* 1500 bytes */


    /*-----------------------------------------------------------------
        Populate Tx fltr template
      -----------------------------------------------------------------*/
```

```
  memset(&tx_fltr, 0, sizeof(ip_filter_type) * NUM_TX_FLTRS);

  tx_fltr[0].ip_vsn = IP_V4;

  tx_fltr[0].ip_hdr.v4.field_mask = IPFLTR_MASK_IP4_NEXT_HDR_PROT |
                                    IPFLTR_MASK_IP4_DST_ADDR;
  tx_fltr[0].ip_hdr.v4.next_hdr_prot = IPPROTO_TCP;
  tx_fltr[0].ip_hdr.v4.dst.addr = dss_htonl(MY_SERVER_ADDR);
  tx_fltr[0].ip_hdr.v4.dst.subnet_mask = dss_htonl(0xFFFFFFFF);

  tx_fltr[0].next_prot_hdr.tcp.field_mask = IPFLTR_MASK_TCP_DST_PORT;
  tx_fltr[0].next_prot_hdr.tcp.dst.port = dss_htons(MY_SERVER_PORT);
  tx_fltr[0].next_prot_hdr.tcp.dst.port = 0;

  qos_spec.tx.fltr_template.num_filters = NUM_TX_FLTRS;
  qos_spec.tx.fltr_template.list_ptr = tx_fltr;

  /*-------------------------------------------------------------------
    Request QoS
  -------------------------------------------------------------------*/
  memset(&qos_request, 0, sizeof(dss_iface_ioctl_qos_request_type));

  qos_request.nethandle = nethandle;
  qos_request.qos = qos_spec;
  qos_request.cback_fn = qos_cback_fn;   /* QoS event notification
   callback */
  qos_request.user_data = NULL;

  if (dss_iface_ioctl(iface_id,
                      DSS_IFACE_IOCTL_QOS_REQUEST,
                      (void *) &qos_request,
                      &dss_errno) != 0)
  {
    MSG_HIGH("QoS request failed", 0, 0, 0);
    return -1;
  }

  rex_clr_sigs(rex_self(), APP_QOS_CB_SIG);
  app_wait(APP_QOS_CB_SIG);

  switch (qos_event)
  {
    case DSS_IFACE_IOCTL_QOS_AVAILABLE_MODIFIED_EV:
```

```
        memset(&qos_flow_spec, 0,
    sizeof(dss_iface_ioctl_qos_get_flow_spec_type));

        qos_flow_spec.handle = qos_request.handle;
        dss_iface_ioctl(iface_id,
                        DSS_IFACE_IOCTL_QOS_GET_GRANTED_FLOW_SPEC,
                        (void *) &qos_flow_spec,
                        &dss_errno)
        /*-------------------------------------------------------------
          Check if granted QoS is ok
        -------------------------------------------------------------*/
        break;

      case DSS_IFACE_IOCTL_QOS_UNAVAILABLE_EV:
        MSG_HIGH("Network didn't grant any QoS", 0, 0, 0);
        return -1;
    }

    /*-----------------------------------------------------------------
      Transmit data and use QoS
    -----------------------------------------------------------------*/
                        .
                        .
                        .

    /*-----------------------------------------------------------------
      Modify QoS
    -----------------------------------------------------------------*/
    memset(&qos_modify_spec, 0, sizeof(qos_spec_type));

    qos_modify_spec.field_mask = QOS_MASK_RX_FLOW;

    /*-----------------------------------------------------------------
      Populate Rx flow template
    -----------------------------------------------------------------*/
    ip_flow_ptr = &(qos_spec.rx.flow_template.req_flow);
    ip_flow_ptr->field_mask = IPFLOW_MASK_TRF_CLASS |
                              IPFLOW_MASK_DATA_RATE;

    ip_flow_ptr->trf_class = IP_TRF_CLASS_STREAMING;

    ip_flow_ptr->data_rate.format_type = DATA_RATE_FORMAT_MIN_MAX_TYPE;
    ip_flow_ptr->data_rate.format.min_max.max_rate = 128000;  /* 128 kbps */
    ip_flow_ptr-> data_rate.format.min_max.guaranteed_rate = 64000; /* 64
  kbps */
```

```
  memset(&qos_modify, 0, sizeof(dss_iface_ioctl_qos_modify_type));

  qos_modify.qos = qos_modify_spec;
  qos_modify.handle = qos_request.handle;

  if (dss_iface_ioctl(iface_id,
                      DSS_IFACE_IOCTL_QOS_MODIFY,
                      (void *) &qos_request,
                      &dss_errno) != 0)
  {
    MSG_HIGH("QoS modify failed", 0, 0, 0);
    /* Release QoS */
    return -1;
  }

  rex_clr_sigs(rex_self(), APP_QOS_CB_SIG);
  app_wait(APP_QOS_CB_SIG);

  switch (qos_event)
  {
    case DSS_IFACE_IOCTL_QOS_MODIFY_ACCEPTED_EV:
      MSG_HIGH("Network accepted QoS modify", 0, 0, 0);
      break;

    case DSS_IFACE_IOCTL_QOS_MODIFY_REJECTED_EV;
      MSG_HIGH("Network rejected QoS modify", 0, 0, 0);
      /* Release QoS */
      return -1;
  }

  /*-------------------------------------------------------------------
    Transmit data and use QoS
  -------------------------------------------------------------------*/
                              .
                              .
                              .

  /*-------------------------------------------------------------------
    Release QoS
  -------------------------------------------------------------------*/
  qos_release.handle = qos_request.handle;
  dss_iface_ioctl(iface_id,
                  DSS_IFACE_IOCTL_QOS_RELEASE,
                  (void *) &qos_request,
```

```
                        &dss_errno) != 0);

    /*-------------------------------------------------------------------
      DSS_IFACE_IOCTL_QOS_UNAVAILABLE_EV is posted. QoS handle is no
       longer valid.
    -------------------------------------------------------------------*/

    rex_clr_sigs(rex_self(), APP_QOS_CB_SIG);
    app_wait(APP_QOS_CB_SIG);
    return 0;
  }

  void qos_cback_fn
  (
    dss_iface_ioctl_event_enum_type        event,
    dss_iface_ioctl_event_info_union_type  event_info,
    void                                   *user_data,
    sint15                                 nethandle,
    dss_iface_id_type                      iface_id
  )
  {
    qos_event = event;
    (void)rex_set_sigs(&app_tcb, APP_QOS_CB_SIG);
  }
```

6.4 Technology specific QoS support

Table 6-2 specifies whether various technologies support assorted QoS features.

Table 6-2 QoS features supported by various technologies

Feature	CDMA	UMTS	WLAN
DSS_IFACE_IOCTL_QOS_REQUEST	Yes	Yes	Yes
DSS_IFACE_IOCTL_QOS_RELEASE	Yes	Yes	Yes
DSS_IFACE_IOCTL_QOS_SUSPEND	Yes	No	No
DSS_IFACE_IOCTL_QOS_RESUME	Yes	Yes	No
Bundled QoS API support (DSS_IFACE_IOCTL_QOS_REQUEST_EX, DSS_IFACE_IOCTL_QOS_RELEASE_EX, DSS_IFACE_IOCTL_QOS_SUSPEND_EX, DSS_IFACE_IOCTL_QOS_RESUME_EX)	Yes	No	No
DSS_IFACE_IOCTL_QOS_MODIFY	Yes	Yes	No
DSS_IFACE_IOCTL_PRIMARY_QOS_MODIFY	No	Yes	No
QoS aware/unaware system support	Yes	No	Yes
Support for verbose parameters in QoS specification	No (profile ID must be used to specify all the parameters)	Yes	Yes
DSS_IFACE_IOCTL_QOS_NET_INITIATED_REQUEST	No	Yes (LTE only)	No

Table 6-3 specifies the limitations pertaining to QoS on various technologies.

Table 6-3 Limitations pertaining to QoS on various technologies

Limitation	CDMA	UMTS	WLAN
Maximum number of QoS requests on an interface	32 (at most 16 in each direction)	2	16
Maximum number of QoS requests per mobile	40	6	16
Support for auxiliary flows in flow specification	Yes	No	No
Maximum number of flows in a flow specification	7	2	2
Maximum number of filters in a filter specification	8	8	8[1]
Maximum number of QoS instances in a bundled QoS API	10	N/A	N/A

In addition to the above, it is not possible to request QoS in only the Tx direction on the UMTS system, i.e., if an application is requesting QoS on a UMTS system, it must request QoS in the Rx direction, at least.

[1] Currently WLAN does not send filters over the air. Therefore, 8 is a soft limit and the limit may change when WLAN sends filters over the air.

7 Multicast Support

The multicast API DSS_IFACE_IOCTL_MCAST_JOIN/LEAVE/ JOIN_EX/LEAVE_EX/REGISTER_EX can be used to join technology-specific multicast groups. The messaging used to join multicast groups is technology-specific, not IGMP, and is specified by the standards for each technology. The IOCTL interface to these technologies, however, is identical.

Currently we support two multicast interfaces, BCMCS and MediaFLO. An application can bind to either of these interfaces by setting its network policy in one of the two ways. If the application specifies the group in its network policy as DSS_IFACE_BCAST_MCAST, the application will receive one of the available broadcast/multicast interfaces, although no particular interface will be guaranteed. If instead the application specifies the particular name of an interface, i.e., DSS_IFACE_CDMA_BCAST (BCMCS), it will attempt to bind the application to that interface. See Section 3.1.2 on dsnet_get_handle() for more information about setting up an application's network policy to request a specific interface.

7.1 Multicast event semantics

7.1.1 Events

When the DSS_IFACE_IOCTL_MCAST_JOIN IOCTL is called it automatically registers the application to receive these three multicast events:

- DSS_IFACE_IOCTL_MCAST_REGISTER_SUCCESS_EV
- DSS_IFACE_IOCTL_MCAST_REGISTER_FAILURE_EV
- DSS_IFACE_IOCTL_MCAST_DEREGISTERED_EV.

When the DSS_IFACE_IOCTL_MCAST_JOIN_EX IOCTL is called, it automatically registers the application to receive four multicast events. The first three events are the same as above, and the fourth event follows:

- DSS_IFACE_IOCTL_MCAST_STATUS_EV

DSS_IFACE_IOCTL_MCAST_REGISTER_SUCCESS_EV indicates that the joining of multicast group on the specified interface was successful. The application must then have a socket bound to the same multicast IP and port to receive the multicast data. When multicast data arrives on the socket the application will get the socket read event indicating that there is data to be read.

DSS_IFACE_IOCTL_MCAST_REGISTER_FAILURE_EV indicates that the group could not be joined. For the BCMCS interface, this may indicate a temporary failure based on the failure code listed in Section 7.1.3. If the failure is permanent, the application should then call dss_iface_ioctl() with the DSS_IFACE_IOCTL_MCAST_LEAVE IOCTL which will relinquish the handle it was assigned and deregister the application for the multicast events.[2]

DSS_IFACE_IOCTL_MCAST_DEREGISTERED_EV indicates the application has been deregistered from the multicast group. The reason for this will be indicated by the accompanying info code. There is no need to call the MCAST_LEAVE IOCTL, as the application has been automatically deregistered. The handle that it was previously assigned is therefore no longer valid.

DSS_IFACE_IOCTL_MCAST_STATUS_EV indicates intermediate status and transient errors occurring during multicast configuration and registration of a flow. The accompanying info code specifies the reason or status.

7.1.2 Duplication of events

Legacy applications share the first three events listed in Section 7.1.1 with the new applications and do not register for the DSS_IFACE_IOCTL_MCAST_STATUS_EV event.

For BCMCS technology, the information codes (see Section 7.1.3) are divided for legacy and new applications. Legacy applications registering through DSS_IFACE_IOCTL_MCAST_JOIN IOCTL will receive events accompanied with information codes having a value less than the DSS_IFACE_IOCTL_MCAST_BCMCS_MAX_DEPRECATED_INFO_CODE.

Backward compatibility is achieved when the shared multicast events (SUCCESS/FAILURE/DEREGISTERED) generated with information codes higher than DSS_IFACE_IOCTL_MCAST_BCMCS_MAX_DEPRECATED_INFO_CODE are duplicated for the legacy applications with appropriate information code values.

Since two instances of similar events will be generated with different information codes, new applications should reject events received with information codes having a value greater than the DSS_IFACE_IOCTL_MCAST_BCMCS_MAX_DEPRECATED_INFO_CODE. See Section 7.1.3 for more details.

7.1.3 Information codes

Information codes will be passed back with the events MCAST_REGISTER_FAILED and MCAST_DEREGISTERED for legacy applications and with all four events for new applications to further indicate to the application the reason for the event. Information codes are technology-specific, indicating for that particular interface the reason for the events. The following list contains the information codes found within dss_iface_ioctl_mcast_info_code_enum_type. Depending upon the requested multicast interface, the application will receive information codes from one of these technology-specific sets.

```
/* Generic info code */
DSS_IFACE_IOCTL_MCAST_IC_NOT_SPECIFIED – The ioctl callback did not specify
an info code.
```

[2] See [Q4] for more information on BCMCS flow status events.

```
/* BCMCS info codes */
DSS_IFACE_IOCTL_BCMCS_FLOW_STATUS_CANCELLED - A new Flow Request received
before the previous one is done processing. Cancel the remaining lookup and
process the new Request (no-op).
DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_STATUS_UNABLE_TO_MONITOR - Flow is
available in overhead but can not be monitored because its channel
assignment conflicts with other flows
DSS_IFACE_IOCTL_BCMCS_FLOW_STATUS_REQUESTED - Flow was registered but not
included in broadcast overhead msg.

DSS_IFACE_IOCTL_BCMCS_FLOW_STATUS_TIMEOUT - Broadcast Supervision Timeout.
The AT is on a system where broadcast is available. The AT timed out
waiting for the flow to, appear in the bcovhd message.
DSS_IFACE_IOCTL_BCMCS_FLOW_STATUS_LOST - The AT lost acquisition and
temporarily disabled its monitored flows. It is treated as if the flows
weren't in the BroadcastOverhead message. Another flow status will be sent
when the AT reacquires.
DSS_IFACE_IOCTL_BCMCS_FLOW_STATUS_SYS_UNAVAILABLE - The AT cannot process
the flow request because the BCMCS protocol is in CLOSED state (HDR not
acquired, etc.)

DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_STATUS_AN_REJECT_NOT_AVAILABLE - Flow was
rejected explicitly by the network, BCMCS Flow ID / BCMCS Program ID not
available
DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_STATUS_AN_REJECT_NOT_TRANSMITTED - Flow
was rejected explicitly by the network, BCMCS Flow ID / BCMCS Program ID
not transmitted.
DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_STATUS_AN_REJECT_INVALID_AUTH_SIG - Flow
was rejected explicitly by the network, Invalid authorization signature
DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_STATUS_NO_MAPPING - The BCDB XML file does
not contain any mapping for the current subnet.
DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_STATUS_ID_NOT_FOUND_FOR_GIVEN_MULTICAST_IP
- The BCDB XML file does not contain a matching FlowID in the current
subnet for the provided Multicast IP.
DSS_IFACE_IOCTL_MCAST_BCMCS_MAX_FLOWS_REACHED - The maximum number of
supported flows supported by the mobile has been reached. There are
currently no more resources available.

DSS_IFACE_IOCTL_MCAST_BCMCS_MAX_DEPRECATED_INFO_CODE - For backward
compatibility.

/*
  BCMCS 2.0: Info codes:
```

```
The info codes indicating transient errors    are duplicated by PS to
STATUS event with the 2p0 info code and to legacy events (SUCCESS/FAILURE)
with the older/deprecated info code. Final errors are duplicated into two
FAILURE events, one with the deprecated event and the other with the 2p0
equivalent. This ensures older/legacy applications receive legacy events
with the older info codes, whereas the events with info coodes higher than
DSS_IFACE_IOCTL_MCAST_BCMCS_MAX_DEPRECATED_INFO_CODE are neglected always.
*/

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_CANCELLED – This is the BCMCS
2.0 equivalent of DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_CANCELLED. This has
been classified as a transient error sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_UNABLE_TO_MONITOR - This is the
BCMCS 2.o equivalent of
DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_UNABLE_TO_MONITOR. This has been
classified as a transient error sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_TIMEOUT - This is the BCMCS 2.0
equivalent of DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_TIMEOUT. This has been
classified as a transient error sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_LOST - This is the BCMCS 2.0
equivalent of DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_LOST. This has been
classified as a transient error sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_SYS_UNAVAILABLE - This is the
BCMCS 2.0 equivalent of DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_SYS_UNAVAILABLE.
This has been classified as a transient error sent with
DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_AN_REJECT_NOT_AVAILABLE - This
is the BCMCS 2.0 equivalent of
DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_AN_REJECT_NOT_AVAILABLE. This has been
classified as a final error sent with DSS_IFACE_IOCTL_REGISTER_FAILURE_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_AN_REJECT_NOT_TRANSMITTED - This
is the BCMCS 2.0 equivalent of
DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_AN_REJECT_NOT_TRANSMITTED. This has been
classified as a final error sent with DSS_IFACE_IOCTL_REGISTER_FAILURE_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_AN_REJECT_INVALID_AUTH_SIG -
This is the BCMCS 2.0 equivalent of
DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_AN_REJECT_INVALID_AUTH_SIG.This has been
classified as a final error sent with DSS_IFACE_IOCTL_REGISTER_FAILURE_EV.
```

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_UNAVAILABLE - This is the BCMCS 2.0 equivalent of DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_UNAVAILABLE. This has been classified as a transient error sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_NO_MAPPING - This is the BCMCS 2.0 equivalent of DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_NO_MAPPING. This has been classified as a transient error sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_ID_NOT_FOUND_FOR_GIVEN_MULTICAST_IP - This is the BCMCS 2.0 equivalent of DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_ID_NOT_FOUND_FOR_GIVEN_MULTICAST_IP. This has been classified as a transient error sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_FLOW_STATUS_REQUESTED - This is the BCMCS 2.0 equivalent of DSS_IFACE_IOCTL_MCAST_FLOW_STATUS_REQUESTED. This has been classified as a transient error sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS2p0_MAX_FLOWS_REACHED - This is the BCMCS 2.0 equivalent of DSS_IFACE_IOCTL_MCAST_MAX_FLOWS_REACHED. This has been classified as a final error sent with DSS_IFACE_IOCTL_REGISTER_FAILURE_EV.

/*
  New Info codes: BCMCS 2.0:
Some new info codes are generated by DS using SUCCESS/FAILURE events, which again are duplicated to SUCCESS/FAILURE/STATUS events. Events posted for newer applications will have the new info code while events for older applications have DSS_IFACE_IOCTL_MCAST_IC_NOT_SPECIFIED as their info code.
 */
DSS_IFACE_IOCTL_MCAST_BCMCS_JOIN_REQ_IN_PROGRESS – Flow registration request was successfully received by DS. This is a transient status sent with DSS_IFACE_IOCTL_STATUS_EV for newer applications and as DSS_IFACE_IOCTL_MCAST_IC_NOT_SPECIFIED with DSS_IFACE_IOCTL_REGISTER_SUCCESS_EV for older ones.

DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_REQUEST_SENT – A success status from DS. Flow request command was successfully posted to DO-CP. This is a transient status sent with DSS_IFACE_IOCTL_STATUS_EV.

DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_STATUS_MAX_MONITORED_FLOWS – This notifies the application that DS has reached the max number of flows that can be monitored. This is a transient status sent with DSS_IFACE_IOCTL_STATUS_EV only.

DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_STATUS_MONITORED – Success status from DS, flow status is successfully monitored. Flow is now active and included in broadcast overhead message. This is a final status sent with DSS_IFACE_IOCTL_REGISTER_SUCCESS_EV for newer applications and as DSS_IFACE_IOCTL_MCAST_IC_NOT_SPECIFIED with DSS_IFACE_IOCTL_REGISTER_SUCCESS_EV for older ones.

DSS_IFACE_IOCTL_MCAST_BCMCS_REGISTRATION_SUCCESS – A transient success status indicating that flow register request command was successfully posted to DO-CP.. This is sent with DSS_IFACE_IOCTL_STATUS_EV for newer applications only.

DSS_IFACE_IOCTL_MCAST_BCMCS_REGISTRATION_NOT_ALLOWED – Registration for the flow cannot be allowed at this time. Application should try registration for this flow again later. This is a transient error event sent with DSS_IFACE_IOCTL_STATUS_EV for newer applications only.

DSS_IFACE_IOCTL_MCAST_BCMCS_REGISTRATION_FAILED – Lower layers attempted to send the registration but the message did not go through. This is indicated through DSS_IFACE_IOCTL_REGISTER_FAILURE_EV for newer applications only.

DSS_IFACE_IOCTL_MCAST_BCMCS_FLOW_DEREGISTERED – Flow has been deregistered from the multicast group and the previously allocated multicast handle is no longer valid.

```
  /* MediaFLO info codes */
DSS_IFACE_IOCTL_MCAST_FLO_IP_OR_PORT_NOT_SUPPORTED
DSS_IFACE_IOCTL_MCAST_FLO_NO_AUTHORIZATION
DSS_IFACE_IOCTL_MCAST_FLO_NO_SYSTEM_COVERAGE
DSS_IFACE_IOCTL_MCAST_FLO_MAX_FLOW_REACHED
```

7.1.4 Multicast JOIN/LEAVE semantics

Upon completion of the multicast join a unique handle will be returned, which can then be used to leave the multicast group and automatically deregister the events. Applications will receive multicast events only for the interface handler they have joined a group on. When an application no longer intends to use a registered multicast flow, it should always call the MCAST_LEAVE IOCTL with the handle it was assigned during JOIN. If an application calls the MCAST_LEAVE IOCTL with a valid handle, it will get a return value of success followed by a DEREGISTERD_EV indicating that the flow is deregistered. The application may choose not to wait for this event. The internal resources will be cleaned up only on this event. If an application calls dsnet_release_handle() without calling the MCAST_LEAVE IOCTL on its flows, all registered flows will be deregistered and resources will be cleaned up. If the application receives the DSS_IFACE_IOCTL_MCAST_DEREGISTERED_EV, the application is already deregistered for that particular handle and the multicast events associated with it. That handle is therefore no longer valid and the application does not have to retain it.

The following steps are necessary to open the network with a multicast interface, join a multicast group, receive multicast data, and leave the group. See Example 12 for the sample code corresponding to these steps.

1. The application must specify a multicast interface when initializing the network policy structure.
2. The application will call dsnet_get_handle() with that policy to get the nethandle.
3. The application must call dsnet_start() with its nethandle to bring up the network.
4. Call DSS_GET_IFACE_ID() to get the iface_id.
5. Call dss_iface_ioctl() with DSS_IFACE_IOCTL_MCAST_JOIN. This will automatically register the application for the multicast events.
6. A non-zero handle will be passed back that is used to identify and leave the multicast group.
7. Open a socket, and call dss_bind() with the multicast ip_addr and the port used while joining the multicast group. The socket binding can occur at any time, but the application will receive the multicast data on this socket only once it has been bound to the multicast IP and port.
8. Once the multicast registration has processed successfully the application will receive the DSS_IFACE_IOCTL_MCAST_REGISTER_SUCCESS_EV.
9. At this point the application can start receiving data on that socket by waiting for the read event and calling dss_recvfrom() or dss_recvmsg().
10. To leave the multicast group, the application must call dss_iface_ioctl() with DSS_IFACE_IOCTL_MCAST_LEAVE and pass the handle originally received during the JOIN call. The application will be automatically deregistered for all multicast events.

Example 14

```
/*-------------------------------------------------------------------
In this example an application joins and leaves a BCMCS multicast session.
-------------------------------------------------------------------*/

dss_iface_ioctl_event_enum_type     mcast_event;
uint32                              mcast_handle;

int join_mcast(void)
{
  sint15                            nethandle, ret, dss_errno, sockfd;
  dss_iface_ioctl_mcast_join_type   mcast_join;
  dss_iface_ioctl_mcast_leave_type  mcast_leave;
  dss_iface_id_type                 iface_id;
  dss_net_policy_info_type          policy_info;
  struct sockaddr_in                localaddr;

  memset(&policy_info, 0 , sizeof(policy_info));
  memset((char *) &localaddr, 0, sizeof(localaddr));
  memset(&mcast_join,  0 , sizeof(dss_iface_ioctl_mcast_join_type));
```

```
1           memset(&mcast_leave, 0 , sizeof(dss_iface_ioctl_mcast_leave_type));
2
3           /*Initialize the policy information to select the BCMCS interface*/
4           dss_init_net_policy_info(&policy_info);
5
6           policy_info.family          = AF_INET;
7           policy_info.policy_flag     = DSS_IFACE_POLICY_SPECIFIED;
8           policy_info.iface.kind      = DSS_IFACE_NAME;
9           policy_info.iface.info.name = DSS_IFACE_CDMA_BCAST;
10
11          /* The application must first bring up the network by calling
12            dsnet_get_handle()with the policy set above. The app must then
13            call dsnet_start() prior to executing the Multicast ioctls
14            below */
15
16          iface_id = dss_get_iface_id( nethandle );
17          if (iface_id == DSS_IFACE_INVALID_ID)
18          {
19            MSG_ERROR("Invalid dss iface id",0,0,0);
20            return -1;
21          }
22
23          sockfd = dss_socket( nethandle,
24                               AF_INET,
25                               SOCK_DGRAM,
26                               IPPROTO_UDP,
27                               &dss_errno);
28          if(sockfd < 0)
29
30          {
31            MSG_ERROR("Error creating socket",0,0,0);
32            return -1;
33          }
34
35          localaddr.sin_family      = AF_INET;
36          localaddr.sin_port        = dss_htons(8600);
37          localaddr.sin_addr.s_addr = dss_htonl(0xE00000B9);
38          /*-----------------------------------------------------------------
39            Now bind the socket to the multicast ip and port that we want to
40            join.
41          -----------------------------------------------------------------*/
42          ret = dss_bind( sockfd,
43                          (struct sockaddr *)&localaddr,
44                          sizeof(localaddr),
```

```
                        &dss_errno
                      );

  mcast_join.nethandle          = nethandle;
  mcast_join.user_data_ptr   = NULL;
  mcast_join.handle          = 0;
  mcast_join.mcast_param_ptr = NULL;
  mcast_join.event_cb        = mcast_cback_fn;

  /* Choose the multicast IP and port to join */
  mcast_join.port            = dss_htons(8600);
  mcast_join.ip_addr.type    = IPV4_ADDR;
  mcast_join.ip_addr.addr.v4 = dss_htonl(0xE00000B9);

  if( dss_iface_ioctl( iface_id,
                       DSS_IFACE_IOCTL_MCAST_JOIN,
                       &mcast_join,
                       &dss_errno) != 0)
  {
    MSG_ERROR("Error in MCAST_JOIN IOCTL",0,0,0);
    return -1;
  }

  rex_clr_sigs(rex_self(), APP_MCAST_CB_SIG);
  app_wait(APP_MCAST_CB_SIG);

  switch (mcast_event)
  {
    case DSS_IFACE_IOCTL_MCAST_REGISTER_SUCCESS_EV:
      MSG_HIGH("Multicast registration success. Start receiving
      data.",0,0,0);
      /* Save the handle for releasing the multicast session */
      mcast_handle = mcast_join.handle;
      break;

    case DSS_IFACE_IOCTL_MCAST_REGISTER_FAILURE_EV:
      MSG_HIGH("Multicast registration failure",0,0,0);
      /* Look at the accompanying info code for registration failure
         cause. Release Multicast Session by calling the MCAST_LEAVE
         IOCTL. */
      return -1;

    case DSS_IFACE_IOCTL_MCAST_DEREGISTERED_EV:
      MSG_HIGH("Registration failed, App already
```

```
            deregistered.",0,0,0);
            /* Look at the accompanying info code for registration failure
               cause. */
            return -1;
          }
          /*-------------------------------------------------------------------
            Register for socket read event. Begin receiving data on the
            multicast socket.
          -------------------------------------------------------------------*/

          /* Leave the multicast group. */
          mcast_leave.nethandle = nethandle;
          mcast_leave.handle = mcast_handle;

          if( dss_iface_ioctl( iface_id,
                               DSS_IFACE_IOCTL_MCAST_LEAVE,
                               &mcast_leave,
                               &dss_errno) != 0)
          {
            MSG_ERROR("Error in MCAST_LEAVE IOCTL",0,0,0);
          }

          return 0;

        } /* join_mcast() */

        void mcast_cback_fn
        (
          dss_iface_ioctl_event_enum_type          event,
          dss_iface_ioctl_event_info_union_type    event_info,
          void                                     *user_data,
          sint15                                   nethandle,
          dss_iface_id_type                        iface_id
        )
        {
          mcast_event = event;
          (void)rex_set_sigs(&app_tcb, APP_MCAST_CB_SIG);
        }
```

7.1.5 Multicast JOIN_EX/LEAVE_EX/REGISTER_EX semantics

Upon completion of the Multicast JOIN_EX, num_flows unique handles will be returned, which can be subsequently used to register or leave the Multicast group. Applications will only receive multicast events for the interface handler their flows have joined a group on. When an application no longer intends to use a registered Multicast flow(s), it should always call the MCAST_LEAVE/LEAVE_EX IOCTL with the handle(s) it was assigned during JOIN_EX. If an application calls the MCAST_LEAVE/LEAVE_EX IOCTL with a valid handle(s), it will get a return value of success followed by a DEREGISTERD_EV (two DEREGISTERED events will be generated to applications using BCMCS technology, one carrying the newly defined info code and the other the deprecated one) for all flows indicating that the flows have been deregistered. The application may choose to not wait for this event. Only on this event will the internal resources be cleaned up. If an application calls dsnet_release_handle() without calling the MCAST_LEAVE/LEAVE_EX IOCTL on all its flows, all registered flows will be deregistered and resources will be cleaned up. If the application receives the DSS_IFACE_IOCTL_MCAST_DEREGISTERED_EV it is already deregistered for that particular handle(s) and the Multicast events associated with it. Those handles are therefore no longer valid and the application need no longer retain them.

The following steps are necessary to open the network with a multicast interface, join a multicast group (optionally register on a multicast group if registration was not performed in the earlier join(JOIN_EX) operation), receive multicast data, and leave the group. See Example 13 for the sample code corresponding to these steps.

1. The application must specify a multicast interface when initializing the network policy structure.
2. Apps will call dsnet_get_handle() with that policy to get the nethandle.
3. Apps must call dsnet_start() with their nethandle to bring up the network.
4. Call DSS_GET_IFACE_ID() to get the iface_id.
5. Call dss_iface_ioctl() with DSS_IFACE_IOCTL_MCAST_JOIN_EX with DSS_IFACE_IOCTL_MCAST_REG_SETUP_NOT_ALLOWED request. This will automatically configure the application's flows in DS, but not register the application's flows to the multicast group.
6. Once the multicast join operation has processed successfully, the application will receive the DSS_IFACE_IOCTL_MCAST_SUCCESS_EV for all registered multicast handles. For applications using BCMCS technology, two events will be received (SUCCESS/STATUS). Newer applications should ignore events with deprecated info codes.
7. Non-zero handles that are used to identify and register for the multicast group will be passed back.
8. Call dss_iface_ioctl() with DSS_IFACE_IOCTL_MCAST_REGISTER_EX with the non-zero handles received in step 7. This will automatically register the application's flows for the multicast events.
9. Open a socket, and call dss_bind() with the multicast ip_addresses and the ports used while joining the multicast group. The socket binding can occur at any time, but the application will

only receive the multicast data on this socket once it has been bound to the multicast IP and the port for that flow.

10. Once the multicast registration has processed successfully the application will receive the DSS_IFACE_IOCTL_MCAST_REGISTER_SUCCESS_EV for all registered multicast handles. For applications using BCMCS technology, two events will be received (either SUCCESS/STATUS or SUCCESS/SUCCESS). Newer applications should ignore events with deprecated information codes. If STATUS events are received, newer applications should ideally wait for a SUCCESS event indicating registration is successful.
11. At this point, the application can start receiving data on the socket mentioned in step 9 by waiting for the read event and calling dss_recvfrom() or dss_recvmsg().
12. To leave the multicast group, the application must call dss_iface_ioctl() with DSS_IFACE_IOCTL_MCAST_LEAVE_EX and pass all the registered handles originally received during the JOIN_EX call. The application will be automatically deregistered for all multicast events for all flows.

Example 15

```
/*-------------------------------------------------------------------
In this example an application joins and leaves a BCMCS multicast session.
-------------------------------------------------------------------*/

dss_iface_ioctl_event_enum_type      mcast_event;
uint32
mcast_handle[DSS_IFACE_MAX_MCAST_FLOWS];

int join_mcast(void)
{
  uint32                                loop, loop_again = 1;
  sint15                                nethandle, ret, dss_errno, sockfd;
  dss_iface_ioctl_mcast_join_ex_type    mcast_join_ex;
  dss_iface_ioctl_mcast_leave_ex_type   mcast_leave_ex;
  dss_iface_id_type                     iface_id;
  dss_net_policy_info_type              policy_info;
  struct sockaddr_in                    localaddr;

  memset(&policy_info, 0 , sizeof(policy_info));
  memset((char *) &localaddr, 0, sizeof(localaddr));
  memset(&mcast_join_ex,  0 , sizeof(dss_iface_ioctl_mcast_join_ex_type));
  memset(&mcast_leave_ex, 0 , sizeof(dss_iface_ioctl_mcast_leave_ex_type));

  /* Initialize the policy information to select the BCMCS
  interface */
  dss_init_net_policy_info(&policy_info);

  policy_info.family          = AF_INET;
  policy_info.policy_flag     = DSS_IFACE_POLICY_SPECIFIED;
  policy_info.iface.kind      = DSS_IFACE_NAME;
  policy_info.iface.info.name = DSS_IFACE_CDMA_BCAST;

  /* The application must first bring up the network by calling
  dsnet_get_handle() with the policy set above. The app must then
  call dsnet_start() prior to executing the Multicast ioctls below */

  iface_id = dss_get_iface_id( nethandle );
  if (iface_id == DSS_IFACE_INVALID_ID)
  {
    MSG_ERROR("Invalid dss iface id",0,0,0);
    return -1;
  }
```

```
1
2           sockfd = dss_socket( nethandle,
3                                AF_INET,
4                                SOCK_DGRAM,
5                                IPPROTO_UDP,
6                                &dss_errno);
7           if(sockfd < 0)
8
9           {
10            MSG_ERROR("Error creating socket",0,0,0);
11            return -1;
12          }
13          localaddr.sin_family      = AF_INET;
14          localaddr.sin_port        = dss_htons(8600);
15          localaddr.sin_addr.s_addr = dss_htonl(0xE00000B9);
16
17          /*-----------------------------------------------------------------
18            Now bind the socket to the multicast ip and port that we want to
19            join.
20          -----------------------------------------------------------------*/
21          ret = dss_bind( sockfd,
22                          (struct sockaddr *)&localaddr,
23                          sizeof(localaddr),
24                          &dss_errno
25                        );
26
27            mcast_join_ex.dss_nethandle          = nethandle;
28            mcast_join_ex.user_data_ptr          = NULL;
29            memset(mcast_join_ex.handle, 0, sizeof(uint32)*
30          DSS_IFACE_MAX_MCAST_FLOWS);
31            memset(mcast_join_ex.mcast_param_ptr, 0, sizeof(void *)*
32          DSS_IFACE_MAX_MCAST_FLOWS);
33            mcast_join_ex.event_cb               = mcast_cback_fn;
34            /* Registartion request turned off for all handles */
35            memcpy(mcast_join_ex.mcast_request_flags, (unsigned char)
36          DSS_IFACE_IOCTL_MCAST_REG_SETUP_NOT_ALLOWED,
37          sizeof(dss_iface_ioctl_mcast_join_ex_req_flags_enum_type) *
38          DSS_IFACE_MAX_MCAST_FLOWS );
39            mcast_join_ex.num_fows = DSS_IFACE_MAX_MCAST_FLOWS;
40
41            /* Choose the multicast IP and port to join */
42            for (loop = 0; loop < DSS_IFACE_MAX_MCAST_FLOWS; loop++)
43            {
44              mcast_join_ex.port[loop]            = dss_htons(8600 + loop);
45              mcast_join.ip_addr[loop].type       = IPV4_ADDR;
```

```
  mcast_join.ip_addr[loop].addr.v4    = dss_htonl(0xE00000B9 +
  loop);
}

if( dss_iface_ioctl( iface_id,
                     DSS_IFACE_IOCTL_MCAST_JOIN_EX,
                     &mcast_join_ex,
                     &dss_errno) != 0)
{
  MSG_ERROR("Error in MCAST_JOIN_EX IOCTL",0,0,0);
  return -1;
}

rex_clr_sigs(rex_self(), APP_MCAST_CB_SIG);
app_wait(APP_MCAST_CB_SIG);

while (loop_again)
{
  switch (mcast_event)
  {
    case DSS_IFACE_IOCTL_MCAST_REGISTER_SUCCESS_EV:
      MSG_HIGH("Multicast configuration success.", 0, 0, 0);
      /* Save the handle for registering the multicast sessions */
      memcpy(mcast_handle, mcast_join_ex.handle,
      sizeof(mcast_handle));
      loop_again = 0;
      break;
    case DSS_IFACE_IOCTL_MCAST_REGISTER_FAILURE_EV:
      MSG_HIGH("Multicast registration failure",0,0,0);
      /* Look at the accompanying info code for registration
     failure cause.
         Release Multicast Session by calling the MCAST_LEAVE
      IOCTL. */
      return -1;

    case DSS_IFACE_IOCTL_MCAST_DEREGISTERED_EV:
      MSG_HIGH("Registration failed, App already deregistered.",
      0, 0, 0);
      /* Look at the accompanying info code for registration
      failure cause. */
      return -1;

    case DSS_IFACE_IOCTL_MCAST_STATUS_EV:
      MSG_HIGH(“Received transient status”, 0, 0, 0);
```

```
            /* Look at the accompanying info code for the transient
            status/error cause or wait for a final event notification
            also before attempting to register the handles. */
            loop_again = 1;
            break;
        } /* end switch */
      } /* end while */

      /* Register all flows to the multicast group. */
      mcast_register_ex.nethandle = nethandle;
      memcpy(mcast_register_ex.handle, mcast_handle,
      sizeof(mcast_handle));
      mcast_register_ex.num_flows = DSS_IFACE_MAX_MCAST_FLOWS;

      if( dss_iface_ioctl( iface_id,
                           DSS_IFACE_IOCTL_MCAST_REGISTER_EX,
                           &mcast_register_ex,
                           &dss_errno) != 0)
      {
        MSG_ERROR("Error in MCAST_REGISTER_EX IOCTL",0,0,0);
        return -1;
      }

      rex_clr_sigs(rex_self(), APP_MCAST_CB_SIG);
      app_wait(APP_MCAST_CB_SIG);

      while (loop_again)
      {
        switch (mcast_event)
        {
          case DSS_IFACE_IOCTL_MCAST_REGISTER_SUCCESS_EV:
            MSG_HIGH("Multicast registration success. Start receiving
            data.", 0, 0, 0);
            loop_again = 0;
            break;

          case DSS_IFACE_IOCTL_MCAST_REGISTER_FAILURE_EV:
            MSG_HIGH("Multicast registration failure",0,0,0);
            /* Look at the accompanying info code for registration
               failure cause. Release Multicast Session by calling the
               MCAST_LEAVE IOCTL. */
            return -1;

          case DSS_IFACE_IOCTL_MCAST_DEREGISTERED_EV:
            MSG_HIGH("Registration failed, App already deregistered.",
            0, 0, 0);
            /* Look at the accompanying info code for registration
            failure cause. */
            return -1;
```

```
            case DSS_IFACE_IOCTL_MCAST_STATUS_EV:
              MSG_HIGH("Received transient status", 0, 0, 0);
              /* Look at the accompanying info code for the transient
                 status/error cause. Ideally wait for a final event
                 notification also before attempting to read on the
                 multicast sockets. */
              loop_again = 1;
              break;
          } /* end switch */
        } /* end while */

        /*-------------------------------------------------------------------
          Register for socket read event. Begin receiving data on the
          multicast socket.
        -------------------------------------------------------------------*/

        /* Leave the multicast group. */
        mcast_leave_ex.nethandle = nethandle;
        memcpy(mcast_leave_ex.handle, mcast_handle, sizeof(mcast_handle));
        mcast_leave_ex.num_flows = DSS_IFACE_MAX_MCAST_FLOWS;

        if( dss_iface_ioctl( iface_id,
                             DSS_IFACE_IOCTL_MCAST_LEAVE_EX,
                             &mcast_leave_ex,
                             &dss_errno) != 0)
        {
          MSG_ERROR("Error in MCAST_LEAVE_EX IOCTL",0,0,0);
          return -1;
        }

        return 0;

      } /* join_mcast() */

      void mcast_cback_fn
      (
        dss_iface_ioctl_event_enum_type          event,
        dss_iface_ioctl_event_info_union_type    event_info,
        void                                     *user_data,
        sint15                                   nethandle,
        dss_iface_id_type                        iface_id
      )
      {
        mcast_event = event;
        (void)rex_set_sigs(&app_tcb, APP_MCAST_CB_SIG);
      }
```

8 IPv6

For more information on supported IPv6 features, see [**Error! Reference source not found.**].

Currently, only the UMTS technology supports IPv6.

8.1 IPv6 privacy extensions

See [**Error! Reference source not found.**], which provides a mechanism for generating private IPv6 addresses that are more difficult to track than public addresses. The intention is that these private addresses will be changed consistently on a shorter periodic basis than the fully qualified *public* address established at the initial connection.

8.1.1 Privacy address generation

A privacy address is created upon an application call to DSS_IFACE_IOCTL_GENERATE_PRIV_IPV6_ADDR. There are two types of private IPv6 addresses supported in the AMSS IPv6 implementation; private shareable and private unique. For both address types there is an *unused timer* that will expire if an application does not bind to the address within two minutes. For shared addresses, if another socket has already bound to the address at some point, the unused timer will not be started. Otherwise, when the unused timer expires, the address will be automatically deleted and the DSS_IFACE_IOCTL_IPV6_PRIV_ADDR_DELETED_EV will be generated to the application. See Example 1 for example code on generating and using IPv6 privacy addresses. See Section 0 for more information on this IOCTL.

8.1.1.1 Private shareable

A private shareable IPv6 address adheres to the rules defined in [**Error! Reference source not found.**] and is shareable among any number of sockets. There can be only one valid private shareable address per interface.

The deprecation lifetime timer for a shareable privacy address will be started when the first socket binds to this address. Subsequent sockets that bind to this address will not affect the timer. The expiration of the timer will cause the deprecation event to be generated (see Section 8.2.3) to applications using that address. It is suggested that applications then request a new shareable address. Once all applications have unbound from this address and the address becomes deprecated, it is deleted.

8.1.1.2 Private unique

A private unique IPv6 address adheres to the rules defined in [**Error! Reference source not found.**], but is unique to a particular application. This address is owned by one particular client and cannot be shared. There can be up to four private unique addresses per interface.

The first bind to this address causes the deprecation lifetime timer to begin. Once the last socket unbinds, the address is deleted, causing the DSS_IFACE_IOCTL_PRIV_ADDR_DELETED_EV to be generated to the application. Expiration of the preferred lifetime timer causes a deprecation event to be generated to the application. It is also suggested that an application request a new private unique address at this point.

8.2 IPv6 event semantics

8.2.1 DSS_IFACE_IOCTL_PREFIX_UPDATE_EV

The DSS_IFACE_IOCTL_PREFIX_UPDATE_EV is used to notify applications when the state of an IPv6 prefix changes. Applications must explicitly register and deregister for this event through the IOCTLs in Sections 0 and 0. The type returned with this event is dss_iface_ioctl_prefix_update_info_type. This event is generated in three different scenarios:

- When a prefix is added
- When a prefix is deprecated
- When a prefix is deleted

Accompanied with the event callback is the prefix, prefix length, and an info code which describes the action taken on the prefix. The corresponding info codes for the above scenarios are:

- DSS_IFACE_IOCTL_PREFIX_ADDED
- DSS_IFACE_IOCTL_PREFIX_DEPRECATED
- DSS_IFACE_IOCTL_PREFIX_REMOVED

If the prefix has been deprecated (preferred lifetime of the prefix expires), indicated by the PREFIX_DEPRECATED info code, no new sockets are able to make connections, although sockets already using the prefix can continue to use it until the valid lifetime expires and the prefix is removed. It is highly recommended that all applications using IPv6 register for this event.

8.2.2 DSS_IFACE_IOCTL_IPV6_PRIV_ADDR_GENERATED_EV

This event is not currently generated. This is for future broadcast interfaces only which must perform DAD and required asynchronous address verification (e.g. WLAN). The type returned with this event is dss_iface_ioctl_priv_addr_info_type.

8.2.3 DSS_IFACE_IOCTL_IPV6_PRIV_ADDR_DEPRECATED_EV

This event is generated when an IPv6 privacy address preferred lifetime timer expires. The address will then become unusable for new sockets. The application should call DSS_IFACE_IOCTL_GET_PRIV_IPV6_ADDR to retrieve a new address, unbind to the current address (see Section 0 for more information on unbinding a socket), and then bind to the new address. Applications are automatically registered for this event once a valid IPv6 privacy address is returned to them. The type returned with this event is dss_iface_ioctl_priv_addr_info_type.

8.2.4 DSS_IFACE_IOCTL_IPV6_PRIV_ADDR_DELETED_EV

This event is generated when an IPv6 privacy address is deleted. The address will then become unusable for all sockets. The application should call DSS_IFACE_IOCTL_GET_PRIV_IPV6_ADDR to retrieve a new address, unbind to the current address (if it is not already unbound, see Section 0 for more information on unbinding a socket), and then bind to the new address. Applications are automatically registered for this event once a valid IPv6 privacy address is returned to them. Applications are automatically deregistered for all privacy events once the IPv6 privacy address is deleted. The type returned with this event is dss_iface_ioctl_priv_addr_info_type.

8.3 UMTS technology support

In UMTS technologies if an application specifies AF_UNSPEC as the IP family type in the policy, the IP family is automatically mapped to AF_INET for IPv4. The only time AF_UNSPEC should be used is with the default profile of 0. An application should not specify AF_UNSPEC and a non-default profile number, since the application should know the IP type of the profile prior to requesting it.

Example 1

```
/*---------------------------------------------------------------------
  In this example an application requests a new IPv6 Privacy address,
  binds a socket and sends data.
---------------------------------------------------------------------*/
int get_priv_ipv6_addr(void)
{
  sint15                                nethandle, ret, dss_errno,
                                        sockfd;
  dss_iface_ioctl_priv_ipv6_addr_type   priv_ipv6_addr;
  dss_iface_ioctl_ev_cb_type            prefix_event;
  dss_iface_id_type                     iface_id;
  dss_net_policy_info_type              policy_info;
  struct sockaddr_in6                   localaddr;

  memset(&policy_info, 0 , sizeof(policy_info));
  memset((char *) &localaddr, 0, sizeof(localaddr));
  memset(&priv_ipv6_addr,  0 ,
  sizeof(dss_iface_ioctl_priv_ipv6_addr_type));
  memset(&prefix_event, 0, sizeof(dss_iface_ioctl_ev_cb_type));

  /* Initialize the policy information to select an IPv6 interface */
  dss_init_net_policy_info(&policy_info);
  policy_info.family          = AF_INET6;

  /* The application must first retrieve a nethandle by calling
     dsnet_get_handle() with the policy set above. */
  iface_id = dss_get_iface_id( nethandle );
```

```
if (iface_id == DSS_IFACE_INVALID_ID)
{
  MSG_ERROR("Invalid dss iface id",0,0,0);
  return -1;
}

/* The application should now register for the PREFIX_UPDATE_EV */
prefix_event.dss_nethandle = nethandle;
prefix_event.event         = DSS_IFACE_IOCTL_PREFIX_UPDATE_EV;
prefix_event.event_cb      = ipv6_cback_fn;
prefix_event.user_data_ptr = appid;
if( dss_iface_ioctl(iface_id,
                    RPC_DSSIOCTL_DSS_IFACE_IOCTL_REG_EVENT_CB,
                    &prefix_event,
                    &dss_errno) != 0)
{
  MSG_ERROR("Error in registering for PREFIX_UPDATE_EV!",0,0,0);
  return -1;
}


/* The app must first bring up the network by calling dsnet_start()
   prior to requesting a new IPv6 privacy address. */
sockfd = dss_socket( nethandle,
                     AF_INET6,
                     SOCK_DGRAM,
                     IPPROTO_UDP,
                     &dss_errno);
if(sockfd < 0)
{
  MSG_ERROR("Error creating socket",0,0,0);
  return -1;
}

/* Initial the IOCTL structure. Since the application would like a
   unique address for its socket is_unique is set to TRUE. */
priv_ipv6_addr.dss_nethandle        = nethandle;
priv_ipv6_addr.iid_params.is_unique = TRUE;
priv_ipv6_addr.event_cb             = ipv6_cback_fn;
priv_ipv6_addr.user_data_ptr        = NULL;
if( dss_iface_ioctl( iface_id,
                     DSS_IFACE_IOCTL_GENERATE_IPV6_PRIV_ADDR,
                     &priv_ipv6_addr,
                     &dss_errno) != 0)
```

```
  {
    MSG_ERROR("Error in GENERATE_IPV6_PRIV_ADDR IOCTL",0,0,0);
    return -1;
  }

  /*-------------------------------------------------------------------
    Now bind the socket to the IPv6 privacy address retrieved from
    the IOCTL call.
  -------------------------------------------------------------------*/
  localaddr.sin_family  = AF_INET6;
  localaddr.sin_port    = dss_htons(8600);
  localaddr.sin6_addr   = priv_ipv6_addr.ip_addr.addr.v6;
  ret = dss_bind( sockfd,
                  (struct sockaddr *)&localaddr,
                  sizeof(localaddr),
                  &dss_errno
                );

  /*-------------------------------------------------------------------
    Register for socket write event. Begin writing data on the IPv6
    socket. When the application wishes to discontinue using the
    address, unbind.
  -------------------------------------------------------------------*/
  ret = dss_bind( sockfd,
                  NULL,
                  0,
                  &dss_errno
                );

  /*-------------------------------------------------------------------
    If the application is done, teardown the network by call
    dsnet_stop() and release the nethandle by calling
    dsnet_release_handle().
  -------------------------------------------------------------------*/

  return 0;
} /* get_priv_ipv6_addr () */

void ipv6_cback_fn
(
  dss_iface_ioctl_event_enum_type          event,
  dss_iface_ioctl_event_info_union_type    event_info,
  void                                     *user_data,
  sint15                                   nethandle,
```

```
  dss_iface_id_type                        iface_id
)
{
  switch (event)
  {
    case DSS_IFACE_IOCTL_PREFIX_UPDATE_EV:
      MSG_HIGH("Prefix update event occurred..",0,0,0);
      /* Check to see if event_info.prefix_info.kind ==
         PREFIX_REMOVED. If so and it matches any privacy address
         currently in use the address has already been deleted. */
      break;
    case DSS_IFACE_IOCTL_IPV6_PRIV_ADDR_GENERATED_EV:
      /* Indicates that for broadcast interfaces which must perform
         DAD(WLAN, etc.) that the asynchronous DAD operation has
         completed. Check event info to see if a valid address has
         been returned if DAD was successful.(Currently
         unsupported) */
      break;
    case DSS_IFACE_IOCTL_IPV6_PRIV_ADDR_DEPRECATED_EV:
      MSG_HIGH("Address is deprecated.",0,0,0);
      /* Look at the accompanying address in the
event_info.priv_ipv6_addr.addr.v6
         If the privacy address is not currently in use, the
         associated socket should unbind from the address and request
         a new one. */
      break;
    case DSS_IFACE_IOCTL_IPV6_PRIV_ADDR_DELETED_EV:
      MSG_HIGH("Address is deleted.",0,0,0);
      /* Look at the accompanying address in the
         event_info.priv_ipv6_addr.addr.v6
         The address accompanying the event is no longer valid. The
         Application should no longer use this address. */
      break;

  }

  /*-------------------------------------------------------------------
    All operations should occur in application context. Queue a
    command to the application, do not perform any significant
    operations in the event callbacks.
  -------------------------------------------------------------------*/
  (void)rex_set_sigs(&app_tcb, APP_IPV6_CB_SIG);
}
```

A References

A.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Application Note: Software Glossary for Customers	CL93-V3077-1
AMSS Support For Internet Protocol Version 6 (IPv6)	80-VD229-1
1xEV-DO Release A Data Over Signaling (DoS)	80-V2510-1
1xEV-DO BCMCS Call Processing FDD	80-V9035-1
Standards	
Data Service Options for Wideband Spread Spectrum Systems	TIA/EIA/IS-707 (Feb 1998)
3GPP Technical Specification	24.008 version 3.5.0 Rel 1999
Resources	
Quinn, B., and D. Schute. Windows® Sockets Network Programming	Addison-Wesley (1996)
Stevens, W.R. TCP/IP Illustrated, Volume I: The Protocols, 1994	Addison-Wesley (1994)
Basic Socket Interface Extensions for IPv6	RFC 2553 (Mar 1999)
Stevens, W.R. UNIX® Network Programming	Prentice-Hall (1990)
Domain names – Implementation and Specification	RFC 1035
Basic Socket Interface Extensions for IPv6	RFC 3493
Internet Control Message Protocol	RFC 792
IPv6 Privacy Extensions	RFC 3041
TCP Selective Acknowledgment Options	RFC 2018
TCP Extensions for High Performance	RFC 1323
DNS Configuration Options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)	RFC 3646