Dialogic
Making Innovation Thrive™

Dialogic® NaturalAccess™ Media Stream Protocol Processing API Developer's Manual

October 2010 64-0501-02 Rev A

www.dialogic.com

Copyright and legal notices

All contents of this document are furnished for informational use only and are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation or its subsidiaries ("Dialogic"). Reasonable effort is made to ensure the accuracy of the information contained in the document. However, Dialogic does not warrant the accuracy of this information and cannot accept responsibility for errors, inaccuracies or omissions that may be contained in this document.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH DIALOGIC® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Due to differing national regulations and approval requirements, certain Dialogic products may be suitable for use only in specific countries, and thus may not function properly in other countries. You are responsible for ensuring that your use of such products occurs only in the countries where such use is suitable. For information on specific products, contact Dialogic Corporation at the address indicated below or on the web at www.dialogic.com.

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.

Any use case(s) shown and/or described herein represent one or more examples of the various ways, scenarios or environments in which Dialogic® products can be used. Such use case(s) are non-limiting and do not represent recommendations of Dialogic as to whether or how to use Dialogic products.

Dialogic, Dialogic Pro, Brooktrout, Diva, Cantata, SnowShore, Eicon, Eicon Networks, NMS Communications, NMS (stylized), Eiconcard, SIPcontrol, Diva ISDN, TruFax, Exnet, EXS, SwitchKit, N20, Making Innovation Thrive, Connecting to Growth, Video is the New Voice, Fusion, Vision, PacketMedia, NaturalAccess, NaturalCallControl, NaturalConference, NaturalFax and Shiva, among others as well as related logos, are either registered trademarks or trademarks of Dialogic Corporation or its subsidiaries. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. The names of actual companies and product mentioned herein are the trademarks of their respective owners.

This document discusses one or more open source products, systems and/or releases. Dialogic is not responsible for your decision to use open source in connection with Dialogic products (including without limitation those referred to herein), nor is Dialogic responsible for any present or future effects such usage might have, including without limitation effects on your products, your business, or your intellectual property rights.

Revision history

Revision	Release date	Notes
9000-62318-10	February, 2002	MVH, Fusion 4.3 Beta
9000-62318-11	April, 2003	MVH, Fusion 4.3
9000-62318-12	December, 2003	MVH, Fusion 4.4
9000-62318-13	December, 2004	MVH, Natural Access 2005-1
9000-62318-14	October, 2005	MVH, Natural Access 2005-1, SP 1
9000-62318-15	December, 2006	LBZ, Natural Access 2005-1, SP 3
9000-62318-16	June, 2007	LBZ, Natural Access 2005-1, SP 4
19000-62318-17	October, 2007	DEH, Natural Access 2005-1, SP 5
9000-62318-18	June, 2008	LBZ, Natural Access R8
9000-62318-19	February, 2009	DEH, Natural Access R8.1
64-0501-01	October, 2009	LBG, NaturalAccess R9.0
64-0501-02 Rev A	October, 2010	LBG, NaturalAccess R9.0.4
Last modified: 2010-10-15		

Refer to www.dialogic.com for product updates and for information about support policies, warranty information, and service offerings.

Table Of Contents

1. Introduction

The *Dialogic® NaturalAccess™ Media Stream Protocol Processing API Developer's Manual* describes how to use the MSPP API to create and destroy media channels between MSPP endpoints at different ends of a Dialogic® NaturalAccess™ Fusion VoIP API gateway. This manual also provides a detailed reference of MSPP endpoint and channel filter parameters, commands, and queries. Use this manual with the *Dialogic® NaturalAccess™ Fusion™ VoIP API Developer's Manual*.

This document defines telephony terms where applicable, but assumes that you are familiar with telephony concepts, circuit switching terminology, and the C programming language.

Terminology

Note: The product to which this document pertains is part of the NMS Communications Platforms business that was sold by NMS Communications Corporation ("NMS") to Dialogic Corporation ("Dialogic") on December 8, 2008. Accordingly, certain terminology relating to the product has been changed. Below is a table indicating both terminology that was formerly associated with the product, as well as the new terminology by which the product is now known. This document is being published during a transition period; therefore, it may be that some of the former terminology will appear within the document, in which case the former terminology should be equated to the new terminology, and vice versa.

Former terminology	Dialogic terminology
CG 6060 Board	Dialogic® CG 6060 PCI Media Board
CG 6060C Board	Dialogic® CG 6060C CompactPCI Media Board
CG 6565 Board	Dialogic® CG 6565 PCI Media Board
CG 6565C Board	Dialogic® CG 6565C CompactPCI Media Board
CG 6565e Board	Dialogic® CG 6565E PCI Express Media Board
CX 2000 Board	Dialogic® CX 2000 PCI Station Interface Board
CX 2000C Board	Dialogic® CX 2000C CompactPCI Station Interface Board
AG 2000 Board	Dialogic® AG 2000 PCI Media Board
AG 2000C Board	Dialogic® AG 2000C CompactPCI Media Board
AG 2000-BRI Board	Dialogic® AG 2000-BRI Media Board
NMS OAM Service	Dialogic® NaturalAccess™ OAM API
NMS OAM System	Dialogic® NaturalAccess™ OAM System

Former terminology	Dialogic terminology
NMS SNMP	Dialogic® NaturalAccess™ SNMP API
Natural Access	Dialogic® NaturalAccess™ Software
Natural Access Service	Dialogic® NaturalAccess™ Service
Fusion	Dialogic® NaturalAccess™ Fusion™ VoIP API
ADI Service	Dialogic® NaturalAccess™ Alliance Device Interface API
CDI Service	Dialogic® NaturalAccess™ CX Device Interface API
Digital Trunk Monitor Service	Dialogic® NaturalAccess™ Digital Trunk Monitoring API
MSPP Service	Dialogic® NaturalAccess™ Media Stream Protocol Processing API
Natural Call Control Service	Dialogic® NaturalAccess™ NaturalCallControl™ API
NMS GR303 and V5 Libraries	Dialogic® NaturalAccess™ GR303 and V5 Libraries
Point-to-Point Switching Service	Dialogic® NaturalAccess™ Point-to-Point Switching API
Switching Service	Dialogic® NaturalAccess™ Switching Interface API
Voice Message Service	Dialogic® NaturalAccess™ Voice Control Element API
NMS CAS for Natural Call Control	Dialogic® NaturalAccess™ CAS API
NMS ISDN	Dialogic® NaturalAccess™ ISDN API
NMS ISDN for Natural Call Control	Dialogic® NaturalAccess™ ISDN API
NMS ISDN Messaging API	Dialogic® NaturalAccess™ ISDN Messaging API
NMS ISDN Supplementary Services	Dialogic® NaturalAccess™ ISDN API Supplementary Services
NMS ISDN Management API	Dialogic® NaturalAccess™ ISDN Management API
NaturalConference Service	Dialogic® NaturalAccess™ NaturalConference™ API

Former terminology	Dialogic terminology
NaturalFax	Dialogic® NaturalAccess™ NaturalFax™ API
SAI Service	Dialogic® NaturalAccess™ Universal Speech Access API
NMS SIP for Natural Call Control	Dialogic® NaturalAccess™ SIP API
NMS RJ-45 interface	Dialogic® MD1 RJ-45 interface
NMS RJ-21 interface	Dialogic® MD1 RJ-21 interface
NMS Mini RJ-21 interface	Dialogic® MD1 Mini RJ-21 interface
NMS Mini RJ-21 to NMS RJ-21 cable	Dialogic® MD1 Mini RJ-21 to MD1 RJ-21 cable
NMS RJ-45 to two 75 ohm BNC splitter cable	Dialogic® MD1 RJ-45 to two 75 ohm BNC splitter cable
NMS signal entry panel	Dialogic® Signal Entry Panel
Video Access Utilities	Dialogic® NaturalAccess™ Video Access Toolkit Utilities
Video Mail Application Demonstration Program	Dialogic® NaturalAccess™ Video Access Toolkit Video Mail Application Demonstration Program
Video Messaging Server Interface	Dialogic® NaturalAccess™ Video Access Toolkit Video Messaging Server Interface
3G-324M Interface	Dialogic® NaturalAccess™ Video Access Toolkit 3G-324M Interface

2. Overview of the MSPP API

MSPP API overview

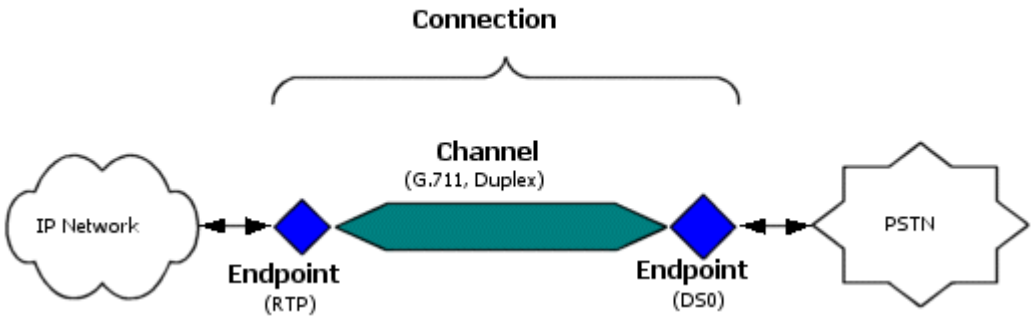
The Media Stream Protocol Processing (MSPP) API provides functions for creating media channels between endpoints established at different ends of a Fusion gateway. The MSPP API enables applications to create unified voice and fax gateway applications by creating, connecting, configuring, and destroying media endpoints and channels. You can use the MSPP API with other NaturalAccess API, such as the ADI and NCC APIs, to provide an interface for PSTN call control and IVR applications.

MSPP connections

The MSPP API provides a mechanism for configuring and managing service components, eliciting runtime query information, and sending runtime commands to MSPP API objects. The MSPP API programming model provides three basic components:

Component	Description
Endpoints	Entry and exit points for data that passes through MSPP channels.
Channels	Linked sets of functions that transform a real-time flow of voice or fax data from one network format to another. Channels can either carry data in two directions (duplex) or one direction (simplex).
Connections	One-way or two-way data streams through the gateway created by associating MSPP endpoints with MSPP channels. When connections are enabled, data flows through the associated channels to the connected endpoints.

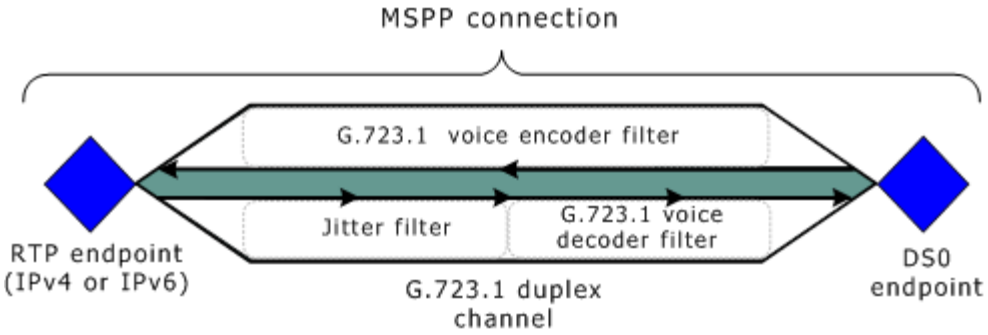
When an application connects an MSPP channel to two MSPP endpoints, it forms a complete MSPP connection. Enabling the connection allows data to flow through the gateway from one network interface to another. The following illustration shows MSPP API connection components:



MSPP filters

MSPP connections consist of linked sets of MSPP filters that transform a real-time flow of voice or fax data from one format to another. An MSPP filter is a subset of operations that transforms the data that flows through an MSPP connection. Endpoints consist of a single filter that translates between a network specific transport format (for example, DS0 or IP format) and a media-specific format (for example, T.38 fax, G.711 voice). A channel consists one or more MSPP filters that perform specific tasks with data that moves between two endpoints.

For example, G.723.1 duplex voice channels use filters to encode and decode (according to the G.723.1 algorithm) a bidirectional flow of data as the data moves between a PSTN and packet network. G.723.1 duplex voice channels also use a jitter filter to perform packet reordering and provide a buffering mechanism for smoothing jitter in the packet stream. Once a connection is created, depending on the characteristics of the filters that make up the connection, the application can command and query individual filters within the connection. In this way, the application can manage and monitor the flow of data through the connection. The following illustration shows MSPP API endpoint and channel filters:



For more information, refer to MSPP endpoint filters or MSPP channel filters.

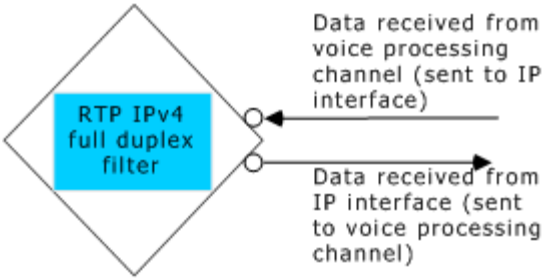
MSPP endpoint filters

Each MSPP endpoint consists of a single MSPP filter. The MSPP service provides the following set of endpoints for sending and receiving data over a network interface:

Endpoint type	Description
DS0 (PSTN)	Entry and exit point for simplex or full duplex PSTN data.
RTP IPv4 full duplex	Entry and exit point for full duplex voice over IPv4 data streams (includes an RTP header within UDP packets).
RTP IPv4 simplex receive	Entry point for simplex voice over IPv4 data streams (includes an RTP header within UDP packets).
RTP IPv4 simplex send	Exit point for simplex voice over IPv4 data streams (includes an RTP header within UDP packets).
RTP IPv6 full duplex	Entry and exit point for simplex voice over IPv6 data streams (includes an RTP header within UDP packets).

Endpoint type	Description
RTP IPv6 simplex receive	Entry point for simplex voice over IPv6 data streams (includes an RTP header within UDP packets).
RTP IPv6 simplex send	Exit point for simplex voice over IPv6 data streams (includes an RTP header within UDP packets).
T38UDP	Entry and exit point for full duplex T.38 fax over UDP data streams (IPv6 not supported).
TPKT	Entry and exit point for full duplex voice data stream multiplexed according to the ThroughPacket algorithm (IPv6 not supported).

The following illustration shows a typical MSPP RTP endpoint. A single RTP full duplex endpoint filter performs all the tasks required to send and receive voice data over the IP interface.



MSPP channel filters

The MSPP service provides the following set of standard channels for processing and conducting data across the gateway:

- Voice encoding and decoding
- T.38 fax
- RTP switching (or pass through)

Each MSPP channel is composed of one or more filters that carry out a set of operations with data that flows through the channel.

Voice encoding and voice decoding channels

The MSPP service provides several filters for encoding and decoding data transferred between PSTN and packet networks according to standard algorithms. These filters include:

- G.723.1/A
- G.711
- G.729A/B
- G.726

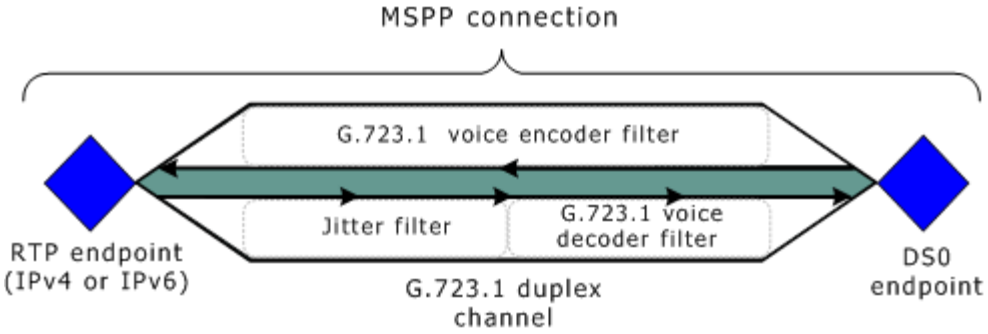
In addition to encoding and decoding the data, MSPP voice encoding/decoding channels perform other operations such as buffering and reordering data frames to provide a steady flow of data to the DS0 endpoint.

The following channel components make up a typical full duplex MSPP voice encoding/decoding channel:

Filter type	Description
Voice decoders	Decode and decompress voice data according to a particular algorithm.
Voice encoders	Encode and compress voice data according to a particular algorithm.
Jitter	Eliminate or reduce jitter in a packet sequence by buffering and rearranging the data frames in the correct sequence.

The following example shows the filters that make up a typical MSPP voice encoding/decoding channel. Although the encoder and decoder filters vary depending on the vocoding algorithm used (for example, G.723.1 or G.729A), the sequence of MSPP filters remains the same for different voice channels.

The following illustration shows the channel filters used in a full duplex G.723.1 voice channel:

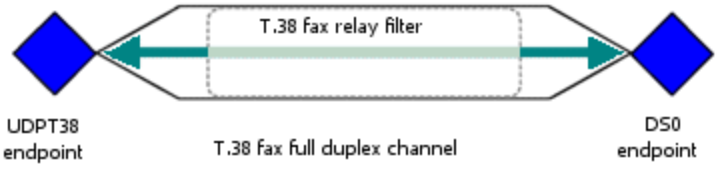


T.38 fax channels

Fusion T.38 fax connections use UDP packets to carry fax data over IP networks. At one end of the IP network a sending gateway converts analog fax data into digital form. It then encapsulates the data within UDP packets with an Internet fax protocol (IFP) header and a UDP transport layer (UDPTL) header, before sending the fax packets over the IP network. Another receiving gateway decodes the UDP fax packets and regenerates the analog fax signals.

MSPP T.38 fax channels use a T.38 fax full duplex filter to convert digital fax data to analog form (modulation) and analog data to digital form (demodulation).

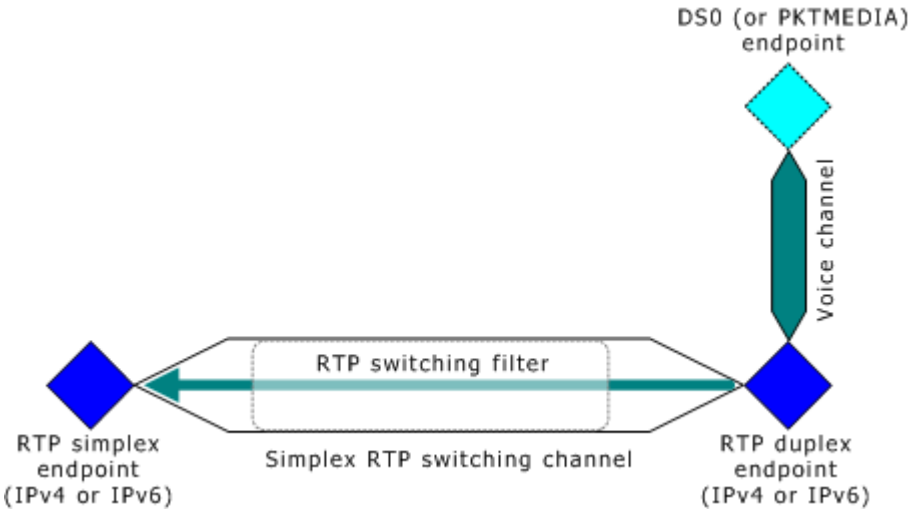
The following illustration shows a Fusion T.38 fax full duplex channel:



RTP switching channels

RTP switching channels act as pass-through filters for simplex data streams traveling between two RTP endpoints. RTP switching filters do not perform any processing tasks with the data that they carry between the RTP endpoints. Applications can connect RTP switching channels to IPv4 endpoints or IPv6 endpoints.

RTP switching channels enable applications to fork and switch RTP data streams to one or more destinations on the IP network. Applications can combine pairs of RTP switching channels (combined with RTP full duplex endpoints) to create a full duplex data path between two RTP full duplex endpoints. The following illustration shows an RTP simplex switching channel:



For more information about RTP switching and forking, refer to the *Fusion Developer's Manual*.

NaturalAccess environment

This topic provides background information about NaturalAccess and summarizes the main elements of the NaturalAccess environment. You must have NaturalAccess installed on your system to build applications using Fusion MSPP functions.

For detailed information about NaturalAccess, refer to the *Dialogic® NaturalAccess™ Software Developer's Manual*.

This topic discusses:

- Synchronous and asynchronous functions
- NaturalAccess OAM API

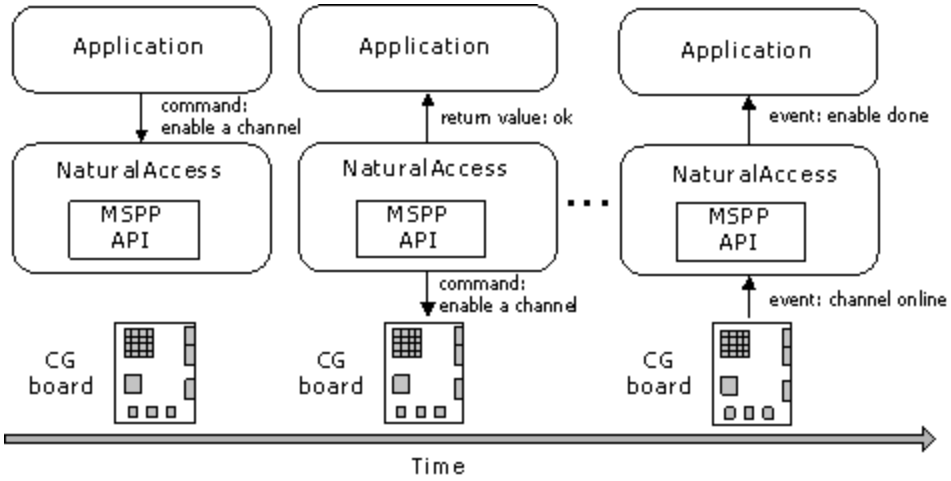
Synchronous and asynchronous functions

NaturalAccess employs an asynchronous programming model to take advantage of concurrent processing. When called, most functions return immediately indicating the operation was initiated. The application can then perform other functions while NaturalAccess is processing the command.

There are two types of functions in NaturalAccess: synchronous and asynchronous.

- Synchronous functions are complete when the return value is received. The return value may be either SUCCESS or an error code.
- Asynchronous functions return events that indicate the success or failure of the operation, and if the operation fails, may return the reason for this failure (in the value field of the event).

As shown in the following illustration, for asynchronous functions, NaturalAccess sends a command to the service that sends a command to a telephony board. The board performs the requested functions and sends events to the service indicating its state (for example, the function was started or the function is complete). The service sends events to NaturalAccess, which makes them available to the application.



NaturalAccess OAM API

The NaturalAccess OAM API provides an API for the OAM software. The OAM API provides tools that applications can use to monitor and manage telephony resources in a system. Use the OAM API to configure and initialize the Dialogic hardware in your chassis. The OAM API can manage:

- Hardware components, such as CG boards.
- Software components, such as the Dialogic Hot Swap process.

Using the OAM API, you can:

- Create, delete, and query a board configuration.
- Start (boot), stop (shut down), and test a board.
- Receive notifications from managed components.

Software processes supported by the OAM API include:

- Hot Swap: the ability to insert and extract CompactPCI boards without powering down the system.
- Clock management: the ability to configure the CT bus clock across all boards in an H.110 system.

For general information about OAM API, refer to the *Dialogic® NaturalAccess™ OAM System Developer's Manual*. For information about the OAM API, refer to the *Dialogic® NaturalAccess™ OAM API Developer's Manual*.

Standards

The following International Telecommunications Union (ITU) specifications and Networking Group RFCs provide detailed information about T.38 facsimile, RTP, and voice media coders:

Document	Description	Date
ITU-T Recommendation G.711	Pulse code modulation (PCM) of voice frequencies	1988
ITU-T Recommendation G.723.1	Dual rate speech coder for multimedia communication transmitting at 5.3 and 6.3 kbit/s	1996
ITU-T Recommendation G.729	Coding of speech at 8 kbit/s using conjugate structure algebraic-code-excited linear-prediction (CSACELP) Annex A: Reduced complexity 8 kbit/s CS-ACELP speech codec	11/96
ITU-T Recommendation G.726	40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM)	12/90
IETF Internet RFC 1889	A Transport Protocol for Real-Time Applications	1/96
IETF Internet RFC 1890	RTP Profile for Audio and Video conferences with Minimal Control	1/96
ITU-T Recommendation T.38	Procedures for Real Time Group 3 Facsimile Communications over IP Networks	6/98
IETF Internet RFC 2833	RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals	5/2000
IETF Internet RFC 2464	A Method for Transmission of IPv6 Packets over Ethernet Networks	12/1998

3. Creating media connections

Media connection overview

The MSPP API enables you to set up and tear down media channels across the gateway system. You can create, connect, and enable duplex or simplex media connections by managing endpoints and channels. MSPP endpoints provide nodes for sending and receiving network data. Channels provide a mechanism for converting and transferring data as it flows from one endpoint to another.

The MSPP API does not perform call control at either the PSTN or packet network interfaces. Call control is performed independently by the application through the Natural Call Control service and a third party call control stack. For information about using the Natural Call Control service, refer to the *Natural Call Control Service Developer's Reference Manual*.

The MSPP API insulates developers from much of the local switching needed to take advantage of CG board DSP resources. Once CG board resource management is configured, you do not need to explicitly perform local switching between the incoming data and the on-board resources needed to process the data. However, you still need to perform local switching between the line interface and the DS0 endpoint.

Setting up the NaturalAccess environment

Before calling functions from the MSPP API, an application initializes NaturalAccess and opens the NaturalAccess services it will use (including MSPP). Applications can use only services initialized with **ctaInitialize**. To set up NaturalAccess:

Step	Action
1	Initialize NaturalAccess and its services.
2	Create event queues and contexts (associated with existing event queues).
3	Opening MSPP API instances on contexts.

To initialize NaturalAccess and its services, invoke **ctaInitialize** and specify the names of the services and service managers the application will use.

The following table lists some of the services and service managers that Fusion applications can use:

This service...	With this service manager...	Provides...
MSP	MSPMGR	Media channel processing and control
ADI	ADIMGR	DTMF detection, tone generation, etc.
NCC	ADIMGR	PSTN call control
SWI	SWIMGR	CT bus switching functions
VCE	VCEMGR	Voice message play and record

For a list of standard NaturalAccess services, refer to the *Dialogic® NaturalAccess™ Software Developer's Manual*.

Creating event queues and contexts

After initializing NaturalAccess and its services, the application does the following:

Step	Action
1	Creates one or more event queues by invoking **ctaCreateQueue**. Specify which service managers to attach to each queue. The MSPP API uses the MSPMGR service manager.
2	Creates contexts by invoking **ctaCreateContext** and providing a queue handle (***ctaqueuehd***) returned from a previous call to **ctaCreateQueue**. Events for any services opened on the context are later routed to the associated event queue.

ctaCreateContext returns a context handle (***ctahd***). The application must specify a valid ***ctahd*** when invoking MSPP functions. Events communicated back to the application are associated with the specified context handle. For more information about NaturalAccess programming models and managing contexts and queues, refer to the *Dialogic® NaturalAccess™ Software Developer's Manual*.

Opening MSPP API instances

ctaOpenServices opens one or more service instances on a particular context. To open an MSPP API instance on a context, invoke **ctaOpenServices** with the following:

- A **ctahd**.
- Service name (MSP) and associated service manager (MSPMGR).
- The board number, stream and timeslot, and mode to associate with the service instance.

For more information about arguments, errors, and events, refer to the *Dialogic® NaturalAccess™ Software Developer's Manual*.

Example: Opening the ADI API and MSPP APIs on a context

In some cases, the application must open multiple services on the contexts associated with MSPP API instances. For example, when an application wants to perform silence detection with the voice data that passes through MSPP connections, it must open an ADI API instance with each context it uses to create DS0 endpoints. To do this, the application must open the MSPP and ADI APIs on the context and specify (in the CTA_MVIP_ADDR structure) the following address and mode information for the ADI API:

CTA_MVIP_ADDR parameter	Value	Description
services[0].mvipaddr.board	NA	Name of the CG board on which to open the service.
svclist.mvipaddr.stream	0	Stream associated with the DSP port. Dialogic recommends using stream 0.
svclist.mvipaddr.timeslot	0 - 1500	Timeslot associated with the DSP port.
svclist.mvipaddr.mode	ADI_VOICE_DUPLEX	Mode of operation for DSP resources allocated to the service instance. For more information about ADI API instances, refer to the *Dialogic® NaturalAccess™ Alliance Device Interface API Developer's Manual*.

Note: When the application uses **mspCreateEndpoint** to create a DS0 endpoint, it must specify the same timeslot (in the DS0_ENDPOINT_ADDR structure) that it specified (in the CTA_MVIP_ADDR substructure) when it opened the ADI API on the context.

Creating MSPP endpoints

An MSPP endpoint provides entry and exit points through which data can flow across a Fusion gateway. Endpoints perform the following operations with the data that passes through them:

- Incoming data: Translate data between a network transport format (for example, RTP or DS0) and the Dialogic media transport format, so that MSPP channels can receive the data, process it, and pass it on.
- Outgoing data: Translate processed data between the Dialogic media transport format and a network transport format so that the data can be transferred to the network.

The MSPP API provides a standard set of endpoints from which to build MSPP connections. Each type of endpoint is capable of sending and receiving data to and from a particular network-specific format.

Available endpoint types include:

Endpoint	Description
DS0	Receive and send data to and from a PCM source (for example, a PSTN).
RTP IPv4	Receive and send data (time stamped with RTP header) to and from the IPv4 network. Endpoints can be simplex or duplex.
RTP IPv6	Receive and send data (time stamped with RTP header) to and from the IPv6 network. Endpoints can be simplex or duplex.
TPKT	Receive and send data to and from an IP network while multiplexing and de-multiplexing packets according to the ThroughPacket algorithm. For more information about ThroughPacket multiplexing, refer to the *Dialogic® NaturalAccess™ Fusion VoIP API Developer's Manual*.
T38UDP	Receive and send data to and from the IP network for T.38 fax channels.

Using mspCreateEndpoint to create endpoints

Applications create MSPP endpoints by invoking **mspCreateEndpoint**. This function requires a ***ctahd*** that has been used to open an MSPP API instance (refer to Opening MSPP API instances).

When using **mspCreateEndpoint**, applications specify the board number on which to create the endpoint, a filter ID that indicates the type of endpoint to create, and the timeslot used to open the MSPP API on the context. The application then specifies an address structure and a parameter structure.

The address structure provides basic configuration information about the endpoint, usually the source and destination address for transferred data. The parameter structure specifies the features that are available on the MSPP endpoint.

The following table provides an overview of the information provided with **mspCreateEndpoint** for different types of MSPP endpoints (RTP IPv4 and RTP IPv6 endpoints are available in simplex and full duplex form):

Endpoint type	Address structure	Parameter structure
DS0	• Board number • Timeslot	• Type of data transmission: voice or fax
RTP IPv4	• Board number • Source UDP port and IPv4 address • Destination UDP port and IPv4 address • Board interface number For simplex endpoints, some of these fields are left NULL because they are inapplicable (for example, the Destination address field is irrelevant for simplex receive endpoints).	• IPv4 type of service information • Number of frames assembled in each packet payload • RTP timestamp frequency • RTCP session parameters • Types of unsolicited events the endpoint returns
RTP IPv6	• Board number • Source UDP port and IPv6 address • Destination UDP port and IPv6 address • Board interface number For simplex endpoints, some of these fields are left NULL because they are inapplicable (for example, the Destination address field is irrelevant for simplex receive endpoints).	• Traffic class of the IP packets transferred through the endpoint • Flow label • Number of frames assembled in each packet payload • RTP timestamp frequency • RTCP session parameters • Types of unsolicited events the endpoint returns
TPKT	• Board number IPv6 is not currently supported.	• Remote and local session ID • Destination gateway IPv4 address • Vocoder type
T38UDP	• Board number • Source port and IPv4 address • Destination port and IPv4 address IPv6 is not currently supported.	• IFP decoder parameters • IFP encoder parameters • UDPTL decoder parameters • UDPTL encoder parameters

When an application creates MSPP endpoints, the MSPP API returns a unique endpoint object handle (***ephd***) that the application can use to configure and control the endpoint.

When an application creates MSPP endpoints with **mspCreateEndpoint**, it uses an address structure to specify the address information associated with the endpoint. The following example shows the DS0_ENDPOINT_ADDR structure:

```
typedef struct tag_DS0_ENDPOINT_ADDR
{
  /* Address attributes */
  INT32           nTimeslot;          /* Timeslot address */
} DS0_ENDPOINT_ADDR;
```

While creating MSPP endpoints, applications also specify parameter structures that define endpoint behavior. The following example shows the DS0_ENDPOINT_PARMS structure:

```
typedef struct tag_DS0_ENDPOINT_PARMS
{
DWORD size
  MSP_MEDIA  media;
} DS0_ENDPOINT_PARMS;
```

The DS0_ENDPOINT_PARMS structure defines whether the DS0 endpoint transfers data in voice or fax mode. Parameter structures for RTP, TPKT, and T38UDP endpoints contain considerably more information.

When creating an endpoint, the application can specify the value 0xFFFF for 16-bit values or 0xFFFFFFFF for 32-bit values (or -1 in decimal format) to accept the default value for any parameters within the endpoint parameter structure. Otherwise, the MSPP API expects the application to provide valid settings for each parameter in the endpoint parameter structure.

When the endpoint is connected, it processes data according to the settings specified when it was created.

For more information, refer to Overview of endpoint structures.

Disabling and enabling MSPP endpoints

By default, when an application creates an endpoint, the endpoint is enabled and ready to transfer information between the network and an MSPP channel. However, the application can use **mspDisableEndpoint** to halt the flow of data through a particular endpoint. Although it usually makes more sense to stop the flow of data by disabling the channel that connects two endpoints, **mspDisableEndpoint** can also be used to interrupt the flow of data through an MSPP connection.

Applications can restart the flow of data through an enabled endpoint by invoking **mspEnableEndpoint**.

Creating MSPP channels

An MSPP channel is made up of one or more filters that transform a real-time flow of voice or fax data from one form to another. For example, an MSPP G.711 duplex channel encodes and decodes (according to the G.711 algorithm) a two-way flow of voice data as it moves between the PSTN and packet network. The application creates the MSPP channels independently of MSPP endpoints, and then uses channels to connect pairs of compatible endpoints.

Applications create MSPP channels with **mspCreateChannel**. When invoking **mspCreateChannel**, the application provides a handle (***ctahd***) and specifies parameters in an address structure and a parameter structure. The MSPP service returns a unique channel object handle (***chnhd***) that the application can use to configure and control the channel. Once the application creates an MSPP channel, it can use the channel to connect MSPP endpoints and create a simplex or duplex media path for voice or fax data.

The following table provides an overview of the information provided with **mspCreateChannel** for different types of MSPP channels:

Channel type	Address structure	Parameter structure
Voice (for example G.711 or G.723.1.)	• Board number • Channel type • Enable or disable special filter attributes (with the FilterAttribs parameter)	• Jitter parameters • Voice encoder parameters • Voice decoder parameters
Record	• Board number • Channel type	Jitter parameters **Note:** Applications specify initial record channel jitter filter settings within an MSP_VOICE_CHANNEL_PARMS structure.
RTP switching	• Board number • Channel type	• None
T.38 fax relay	• Board number • Channel type	• Enable or disable channel features (NSF filtering or ECM) • Types of unsolicited events the endpoint returns • Maximum bit rate and timeout value for the fax transmission

The application specifies the type of channel it wants to create (voice, T.38 fax or switch) in the channel address structure shown below:

```
typedef struct tag_MSP_CHANNEL_ADDR
{
    DWORD            size;
    DWORD            nBoard;
    MSP_CHANNEL_TYPE channelType;
    DWORD            FilterAttribs;
} MSP_CHANNEL_ADDR;
```

Each MSPP channel is composed of a linked set of filters that are organized to carry out operations with data that flows through the channel. When the application creates the channel type in the channel address structure, the MSPP service initializes all the filters necessary to carry out the operations performed by that channel.

The MSPP service provides the following channel types (voice channels are supported in duplex and simplex form):

Channel	Description
Voice duplex	Transfers encoded voice data to an IP network (in G.711, G.723.1, G.729A, or G.726 format) and decoded voice data to a PSTN.
Voice encoder	Transfers encoded voice data to an IP network (in G.711, G.723.1, G.729A, or G.726 format).
Voice decoder	Transfers decoded from G.711, G.723.1, G.729A, or G.726 format to voice data for a PSTN.
Record	Performs jitter functions for native record audio streams (in G.711, G.723.1, G.729A, or G.726 format) without performing voice encoding or decoding.
RTP switching	Transfers a simplex data stream between RTP endpoints (IPv4, IPv6, or a combination).
T.38 fax	Transfers a duplex T.38 fax data stream between T38UDP and DS0 endpoints and converts digital fax data to analog form and analog data to digital form.

When using **mspCreateChannel**, the application also provides a channel parameter structure that configures specific elements of the channel's functionality (the parameter structure is not required for RTP switching channels). For example, when creating a T.38 fax full duplex channel the application specifies the following parameters:

```
typedef struct tag_msp_FILTER_FAXRELAY_CONFIG
{
  WORD  modemask;
  WORD  eventmask;
  WORD  maxbitrate;
  WORD  timeout;
} msp_FILTER_FAXRELAY_CONFIG;
```

Parameters in the msp_FILTER_FAXRELAY_CONFIG structure specify various aspects of the fax relay filter's behavior such as the event mask associated with the filter and the maximum bit rate of fax transmissions processed by the filter.

When creating a channel, the application can specify the value 0xFFFF for 166-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to accept the default value for any parameters within the channel's parameter structure. Otherwise, the MSPP service expects the application to provide valid settings for each parameter in the channel parameter structure.

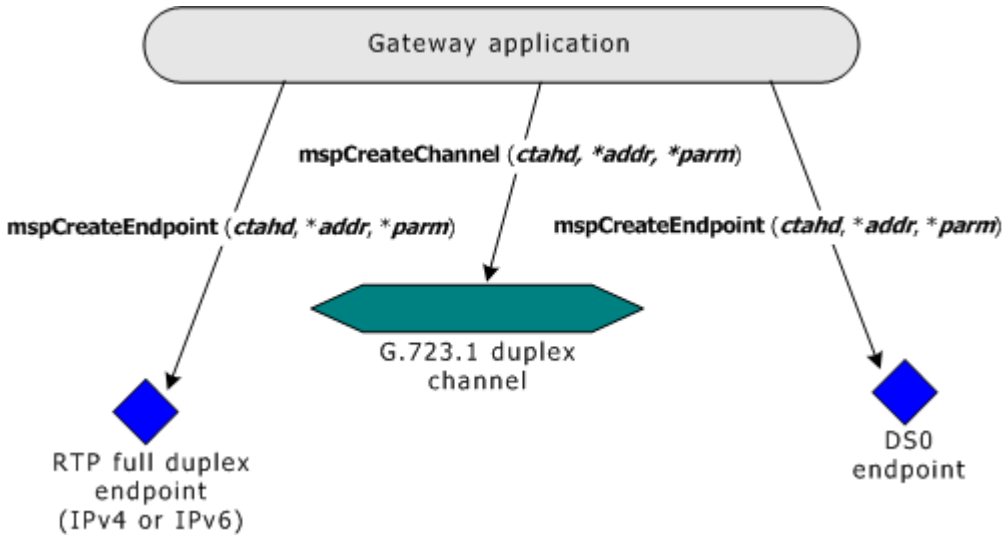
Connecting MSPP endpoints with channels

Applications use **mspConnect** to join MSPP channels with pairs of MSPP endpoints to create simplex or duplex media paths through the system. These end-to-end gateway media paths are called MSPP connections. When invoking **mspConnect**, applications specify two endpoint handles (returned by **mspCreateEndpoint**) and a channel handle (returned by **mspCreateChannel**). These handles identify the endpoints through which the data flows, and the processing that takes place (such as, encoding and decoding) as data flows through a channel.

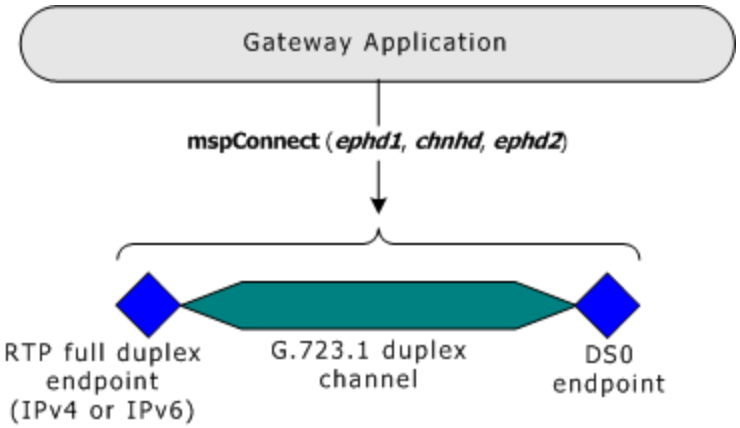
The following table shows the types of MSPP endpoints that can be connected to particular MSPP channels. The first endpoint listed in each endpoint combination-pair is always ***ephd1***, and the second endpoint listed is ***ephd2***:

Channel type	Endpoint combinations (*ephd1*/*ephd2*)
Voice duplex (G.711, G.723.1, G.726, or G.729A)	• RTP and DS0 • TPKT and DS0
Voice encoder (G.711, G.723.1, G.726, or G.729A)	• RTP and DS0 • TPKT and DS0
Voice decoder (G.711, G.723.1, G.726, or G.729A)	• RTP and DS0 • TPKT and DS0
Record (G.711, G.723.1, G.726, or G.729A)	RTP and MSP_NO_CONNECT
RTP switching (simplex and duplex)	RTP and RTP
T.38 fax	T38UDP and DS0

When applications use **mspConnect**, the order in which they specify endpoint handles is significant. For voice channels, application must specify an RTP endpoint handle (ephd1) first, a voice channel handle (chnhd2), and then a DS0 endpoint handle (ephd2). The following example shows an application using **mspCreateChannel** and **mspCreateEndpoint** to create a G.723.1 channel, an RTP endpoint, and a DS0 endpoint:



The following example shows the application using **mspConnect** to connect the RTP endpoint with the DS0 endpoint by means of the G.723.1 channel, thus creating a G.723.1 duplex connection:



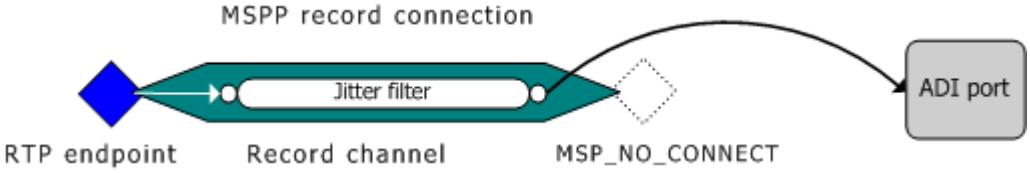
Once the application establishes the connection, the connection begins processing data as soon as the application completes the following tasks:

1. Enables the channel.
2. Connects data to one of the endpoints that make up the connection.

By default, MSPP endpoints are enabled when created, while MSPP channels are disabled when created. For this reason, channels have to be enabled to transfer data, while endpoints (unless explicitly disabled) automatically transfer any data connected to them.

Native record connections

To create a native record channel that doesn't implement inband silence detection or DTMF detection, the application specifies an RTP endpoint, a record channel (that consists of a single jitter filter) and specifies MSP_NO_CONNECT instead of a second endpoint handle. The resulting connection consists of a record channel and one endpoint. Applications can use record connections to transfer un-decoded audio data directly from an RTP endpoint to an ADI port. The following illustration shows a native record connection without voice decoding:



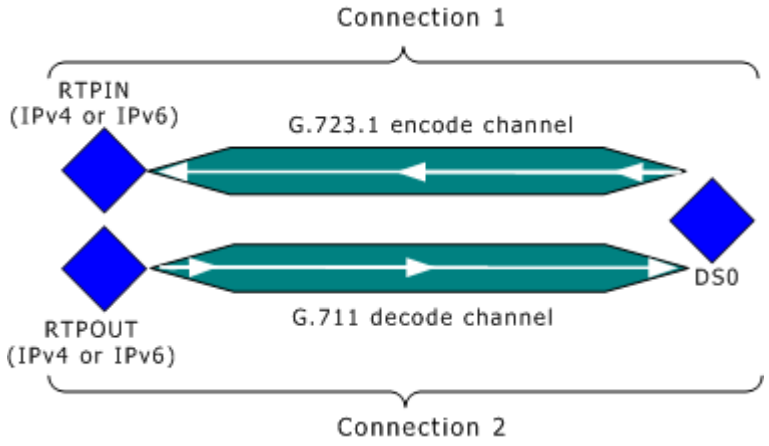
For more information about implementing Dialogic play and record, refer to the *Dialogic® NaturalAccess™ Fusion VoIP API Developer's Manual*.

Connecting simplex and duplex channels

Applications create end-to-end media channels by joining pairs of endpoints with simplex channels or duplex channels to the same endpoint. You can configure the flow of data through a connection in the following ways:

Connection	Description
Simplex	Create simplex connections by creating simplex channels (encode or decode) and connect them with the appropriate endpoints. The connection carries data in a single direction.
Symmetrical duplex	Create duplex connections by creating duplex channels of the same type (encode and decode) and connect them with the appropriate endpoints. The connection carries data in two directions.
Asymmetrical duplex	Create a simplex encode channel of one type (for example, G.711 or G.723), a decode channel of another type (for example G.729A or G.726), and connect them both to the same endpoints. Each connection carries data in a different direction (and possibly encoded according to a different algorithm), but together the connections make up a full duplex data path.
RTP multiple unicast	Create a series of RTP simplex send endpoints (up to eight) and connect all of the these RTP endpoints to a DS0 endpoint through a voice encoder channel.

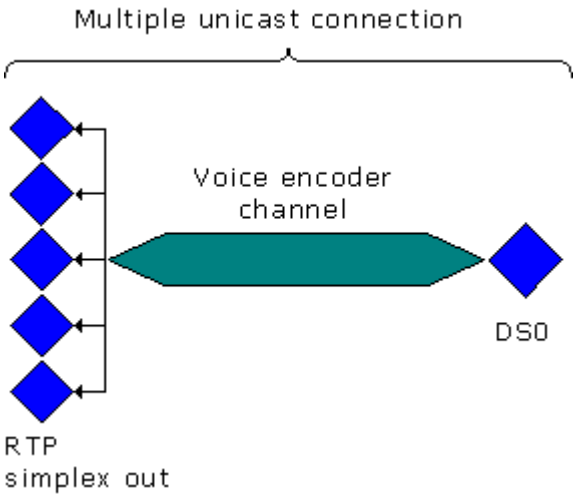
For example, in the following illustration the application creates an asymmetric duplex connection between RTP simplex endpoints (IPv4 or IPv6) and a DS0 endpoint by creating a simplex G.723.1 encode connection in one direction, and a simplex G.711 decode connection in the opposite direction. Since both connections share the same endpoints, data flowing across the channel is encoded and/or decoded according to a different algorithm, depending on the direction in which it flows. The following illustration shows a pair of asymmetrical MSPP simplex connections:



Using RTP multiple unicast connections

RTP multiple unicast connections enable applications to transfer an encoded simplex stream of voice data (through a simplex voice encoder channel) to multiple RTP simplex endpoints. Applications can connect up to eight RTP simplex endpoints to one DS0 endpoint through a single simplex voice encoder channel.

The following illustration shows an RTP multiple unicast connection:



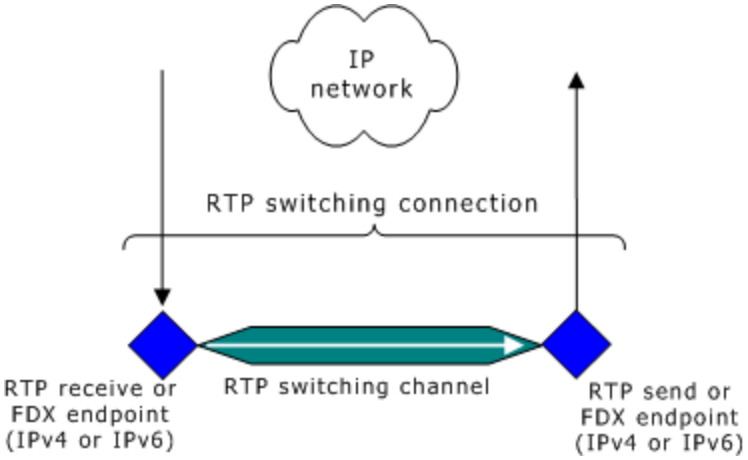
Applications create RTP multiple unicast connections in the following way:

Step	Action
1	Create the required number of RTP simplex send (MSP_ENDPOINT_RTPOUT or MSP_ENDPOINT_RTPOUT_V6) endpoints with **mspCreateEndpoint**.
2	Create a single DS0 endpoint with **mspCreateEndpoint**.
3	Create a single simplex voice encoder channel with **mspCreateChannel**.
4	Invoke **mspConnect** once for each RTP simplex endpoint, specifying the same simplex encoder channel created in Step 3, and the same DS0 endpoint created in Step 2.
5	Enable simplex encoder channel with **mspEnableChannel**.

To dismantle an RTP multiple unicast connection, disable the encoder channel with **mspDisableChannel**. Then invoke **mspDisconnect** for each RTP simplex send endpoint before destroying the associated DS0 endpoint, the RTP endpoints, or the encoder channel.

Connecting RTP switching channels

Applications can also use switch channels to route voice data to more than one RTP (IPv4 or IPv6) endpoint. This mechanism allows applications to monitor the voice data transferred across a particular voice channel. The following illustration shows the components used to connect voice data to multiple RTP endpoints.



When creating switching connections, the RTP endpoint that receives the incoming data must be either an RTP full duplex or RTP receive endpoint. The RTP endpoint that transfers the data back to the network must be either an RTP full duplex or RTP send endpoint. Because switching channels are always simplex (unlike voice channels which can be simplex or duplex), applications must connect two switching channels to create a full duplex connection between RTP endpoints. For more information about RTP switching connections refer to the *Dialogic® NaturalAccess™ Fusion VoIP API Developer's Manual*.

Enabling MSPP connections

MSPP channels are disabled when they are created. After the application establishes a connection by joining MSPP endpoints with an MSPP channel, the application can enable the connection with **mspEnableChannel**. **mspEnableChannel** opens the channel so that data flows between the connected endpoints. When invoking **mspEnableChannel**, specify the channel handle (***chnhd***) of the connected channel. As soon as the application transfers data to one of the connection's endpoints, the connection channel processes the data and passes it on to the endpoint at the opposite end of the connection.

Sending MSPP filter commands and queries

This topic describes:

- Sending endpoint and channel filter commands that modify the component's parameters and effect the component's behavior.
- Releasing event buffers allocated during the filter command process.
- Sending endpoint and channel filter queries that return information about the component's current configuration or activity.

Sending endpoint and channel filter commands

MSPP channels and endpoints are composed of linked sets of processes, called filters. Each filter performs a specific task or set of tasks with the data that flows through it. While all MSPP endpoints consist of a single endpoint filter, most MSPP channels contain multiple channel filters that are used to process data as it moves from endpoint to endpoint.

Applications control MSPP components by using **mspSendCommand** to convey a filter-specific command to a particular endpoint or channel filter. When invoking **mspSendCommand**, the application specifies:

- A valid MSPP ***ephd*** or ***chnhd*** associated with the endpoint or channel filter
- A valid filter ID concatenated with a valid command ID
- A buffer containing the filter command structure
- The size of the command structure

To build an MSPP filter command, use the macro **mspBuildCommand** (defined in *mspcmd.h*) to concatenate a valid MSPP filter ID with a command ID. For more information about sending commands to endpoints and channels, refer to Sending commands to endpoint filters and Sending commands to channel filters.

Releasing event buffers

mspSendCommand returns a DONE event that specifies the success or failure of the filter command. If any data is returned in the MSPEVN_SENDCOMMAND_DONE event, the event.buffer field contains a pointer to the data and the event.size field specifies the size of the returned buffer. After receiving MSPEVN_SENDCOMMAND_DONE, applications must free the associated buffer with **mspReleaseBuffer**.

Sending endpoint and channel filter queries

The MSPP API provides a set of standardized queries that applications can use to solicit configuration and operation data from MSPP filters. MSPP queries to MSPP objects return filter status information in a buffer that must be released by **mspReleaseBuffer**.

To send a query to a filter within a connected channel, the application can invoke **mspSendQuery** and provide:

- An ***ephd*** or ***chnhd***
- A filter ID concatenated with a query ID

To build an MSPP filter query, use the macro **mspBuildCommand** (defined in *mspcmd.h*) to concatenate a valid MSPP filter ID with a query ID. The MSPP API returns a filter-specific structure that provides information about the filter's configuration and status. The same restrictions that apply to sending commands to MSPP components (described in Sending queries to MSPP filters) also apply to sending component queries.

When to send filter commands and queries

Applications can send commands or queries to any MSPP API endpoint or channel filter. However, filters can receive commands and events only in particular states.

The following table shows the states in which MSPP endpoint filters can receive filter commands and queries:

Filter	Command state	Query state	Exceptions
DS0	Created	Not applicable	
RTP IPv4 (full duplex, send, and receive)	Created	Created	MSP_CMD_RTPFDX_CONFIG commands can be sent only when the endpoint is disabled.
RTP IPv6, (full duplex, send, and receive)	Created	Created	MSP_CMD_RTPFDX_V6_CONFIG commands can be sent only when the endpoint is disabled.
TPKT	Disabled	Created	
T38UDP	Created	Created	MSP_CMD_T38UDP_CONFIG commands can be sent only when the endpoint is disabled.

The following table shows the states in which MSPP channel filters can receive filter commands and queries:

Filter	Command state	Query state
Voice decoder	Enabled	Enabled
Voice decoder	Enabled	Enabled
Jitter	Created	Created
RTP switching	Not applicable	Connected
T.38 fax relay	Enabled	Enabled

Disconnecting and destroying MSPP connections

Use the following functions to destroy MSPP connections:

Function	Description
mspDisableChannel	Interrupts the flow of data on a specified MSPP connection.
mspDisconnect	Disassociates the MSPP endpoints and MSPP channel in a specified connection (if the channel is enabled when the command is issued, the MSPP service automatically disables the connection).
mspDestroyEndpoint	Destroys a specified MSPP endpoint (and disables and disconnects the associated connection if the endpoint is part of an enabled connection).
mspDestroyChannel	Destroys a specified MSPP channel (and disables and disconnects the associated connection if the channel is part of an enabled connection).

The order in which the functions are called is should important and must match the order listed in the previous table.

MSPP function sequence and state machine

This topic provides the following illustrations:

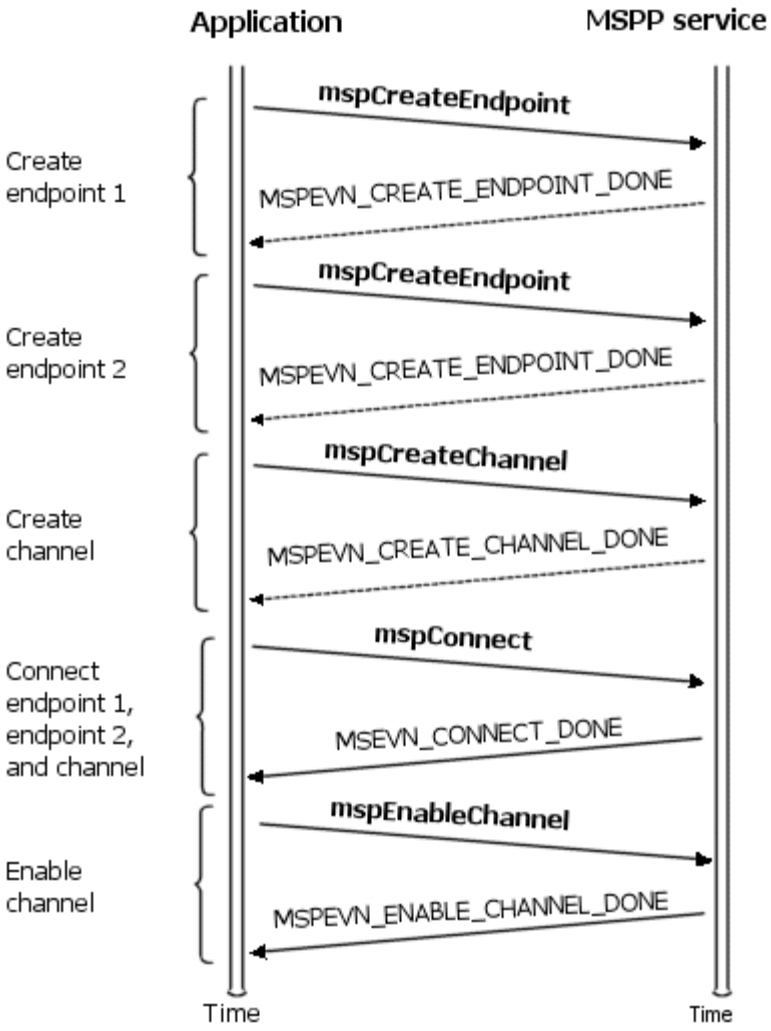
- MSPP function setup and tear down sequence
- MSPP state machine

When setting up and tearing down MSPP connections, keep the following in mind:

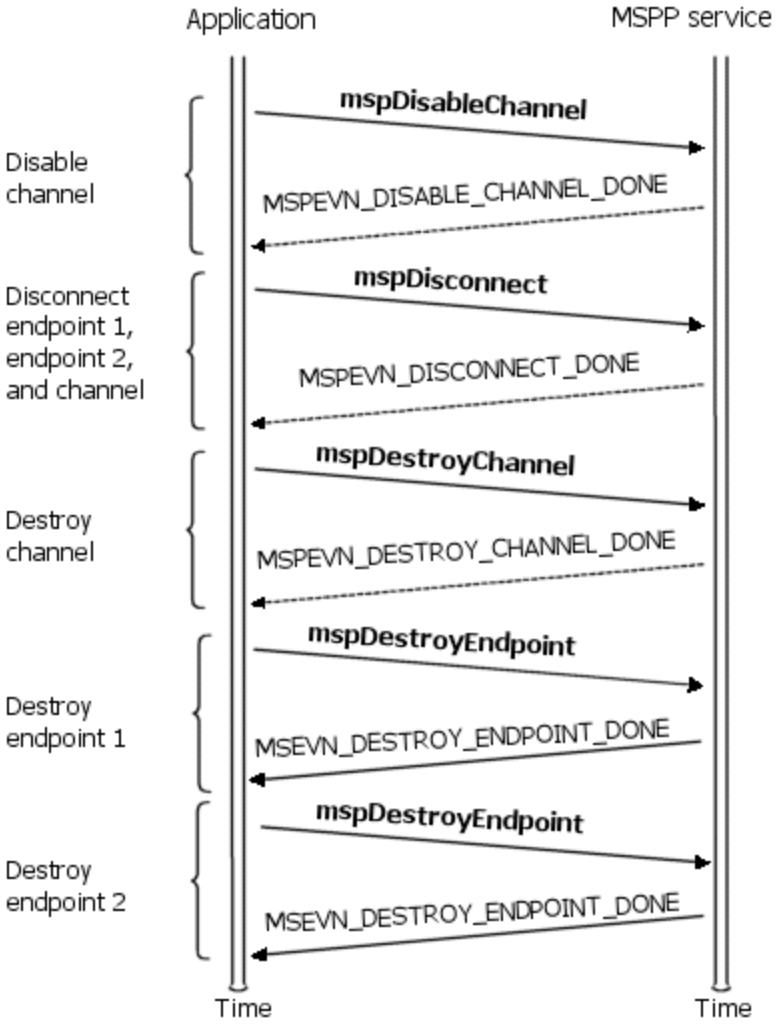
- The application can create endpoints and channels in any sequence.
- Applications can interrupt a connection either by disabling the channel (with **mspDisableChannel**) or by disabling one of its component connection endpoints (with **mspDisableEndpoint**).
- Channel filters cannot receive commands until the channel is connected.

MSPP function setup and tear down sequence

The following illustration shows a typical MSPP function sequence for setting up MSPP connections.

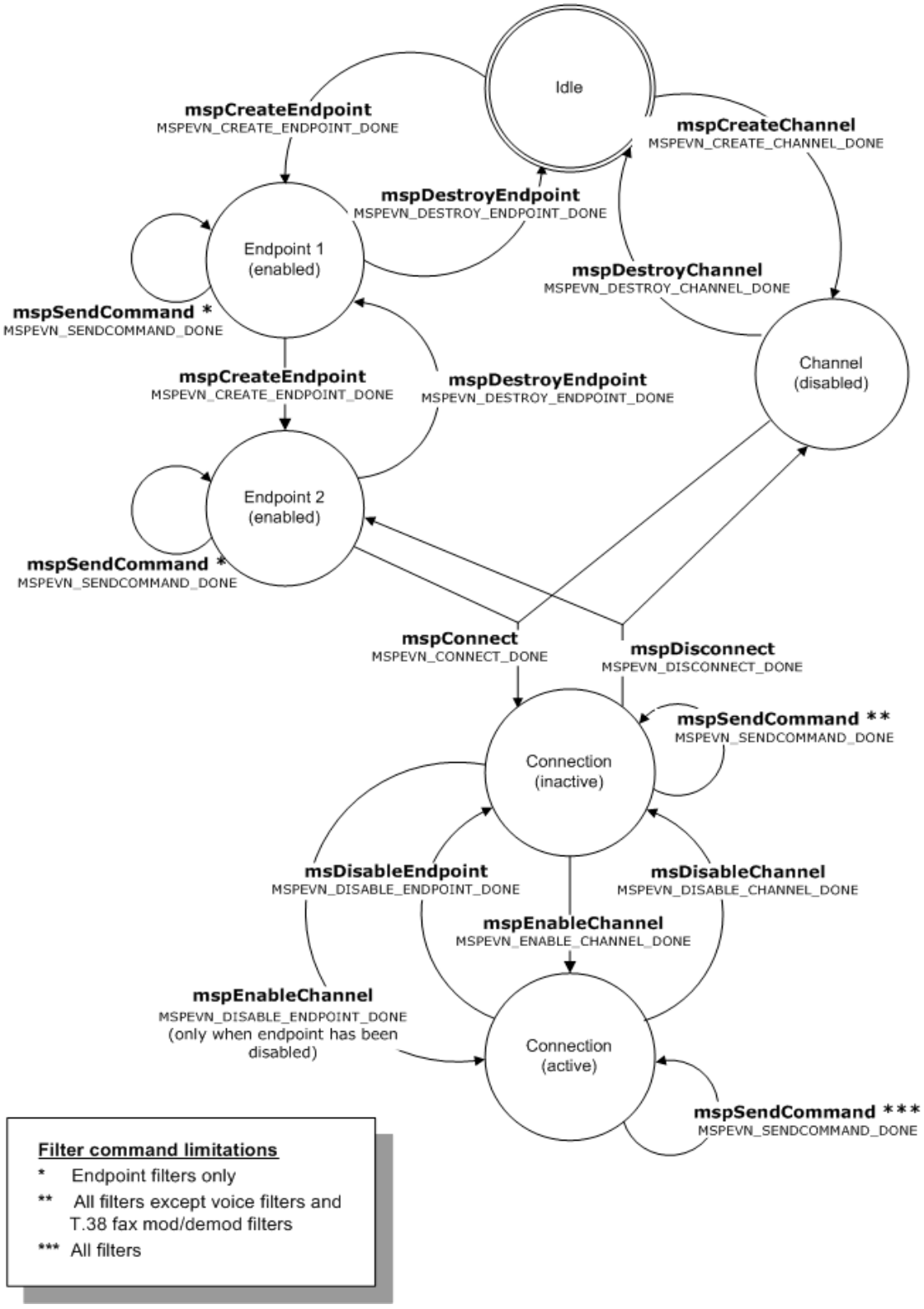


The following illustration shows a typical MSPP function sequence for tearing down MSPP connections.



MSPP state machine

The following illustration shows is a state machine for creating and destroying an MSPP connection.



MSPP unsolicited events

The MSPP service returns unsolicited events to the application when a channel or endpoint detects a particular condition. These events are not responses to specific filter commands and they can be returned at any time while a channel or endpoint is active. Depending on the filter, the endpoint can automatically send unsolicited events when it is created, or may need to be configured or commanded to send the unsolicited events.

For example, when an application creates a T38UDP fax endpoint (endpoints are enabled by default when created), the endpoint automatically begins monitoring incoming data to detect any T.38 packets. The first time it detects a T.38 fax packet, it sends an unsolicited event to the application.

However, RTP endpoint filters do not return unsolicited events with RTCP monitoring information unless the application explicitly turns on this monitoring. The application initiates RTCP monitoring by setting parameter values in the RTPRTCP_ENDPOINT_PARMS structure when creating the endpoint.

After receiving MSPP service unsolicited events, applications must free the associated buffer with **mspReleaseBuffer**. However, calling **mspReleaseBuffer** twice for the same buffer creates an access violation.

The following table lists unsolicited events returned by MSPP filters:

Event	Associated filter	Description
MSPEVN_RTCP_REPORT	RTP (IPv4 and IPv6) duplex	Provides RTCP information for a particular voice connection. Useful for monitoring packet quality of service (QOS). These are disabled by default.
MSPEVN_DPF_DTMF_PLAY_END	Voice decoder filter	Notifies the application that the voice decoder filter has finished playing a DTMF tones as a result of an MSP_CMD_PLAY_DTMF_DIGIT filter command.
MSPEVN_DPF_DTMF_REPORT	Voice encoder filter	Notifies the application when the voice encoder filter detects the presence of a DTMF input signal.
MSPEVN_RFC2833_REPORT	RTP duplex and RTP receive (IPv4 and IPv6) endpoints	Notifies the application that the RTP endpoint received an RFC 2833 RTP packet.
MSPEVN_FAXRELAY_FAX_BEGIN MSPEVN_FAXRELAY_FAX_END MSPEVN_FAXRELAY_PASSED_PAGE	T.38 fax channel filter	Returns status information about an active T.38 fax session.

Event	Associated filter	Description
MSPEVN_T38_PACKET_DETECTED	T38UDP endpoint	Notifies the application that a T38UDP endpoint received a T.38 fax packet (that is, voice fax switchover can be initiated).
MSPEVN_NOROUTE_AVAILABLE	Not applicable. This event returns board-level information and therefore is not associated with a specific filter.	Indicates that an RTP or T38UDP endpoint has lost the ability to route outbound data to its destination.
MSPEVN_ROUTE_AVAILABLE	Not applicable. This event returns board-level information and therefore is not associated with a specific filter.	Indicates that an RTP or T38UDP endpoint lost the ability to route outbound data to its destination.

For more information about scenarios for receiving unsolicited events, refer to the *Fusion Developer's Manual*.

4. Working with MSPP filters

Sending commands to endpoint filters

Applications can control MSPP endpoints by using **mspSendCommand** to convey filter-specific commands to endpoint filters. When sending endpoint filter commands, the application must specify:

- A unique MSPP endpoint handle (obtained when creating the endpoint with **mspCreateEndpoint**).
- A valid filter command produced by using the OR operator to combine a filter ID with a filter command ID.

  Use the macro **mspBuildCommand** (defined in *mspcmd.h*). This macro concatenates the endpoint filter ID with an endpoint command.

  mspBuildCommand is defined in the following way:

```
/* Macro used to build commands with filter definitions */

#define mspBuildCommand(filter,command) ((filter << 8) | command)
```

- A pointer to a command structure that contains the value to assign for the command.
- The size of the command structure.

MSPEVN_SENDCOMMAND_DONE returns the modified ENDPOINT_PARMS structure in a buffer, including any entries modified by the associated filter command. Applications must wait for MSPEVN_SENDCOMMAND_DONE before sending a subsequent filter command to the same endpoint.

The commands available for each type of MSPP endpoint filter are listed in the Channel filter reference and in the *mspcmd.h* header file. RTP (Ipv4 and IPv6) and TPKT endpoints must be disabled before they can receive filter commands. DS0 endpoints can receive filter commands at any time.

Example: Adjusting the outgoing packet frame count

Fusion gateway applications can change the number of voice frames included in RTP packet payloads. The following sample code shows how an application uses **mspSendCommand** to change the frame quota for packets transferred by an RTP IPv4 full duplex endpoint:

```
     do
      {
        printf("Please enter a value between 1 and 6\n");
        scanf("%hd",&nFramesPerPkt);

      }while( (nFramesPerPkt > 6) || (nFramesPerPkt <0) ) ;
       pRtpAssemblerBuffer->value = H2NMS_DWORD(nFramesPerPkt);

       MspOutHd = &pVoiceChannelDbase[index]->hRtpRtcpFdxMspHd;
       CtaOutHd = &pVoiceChannelDbase[index]->hRtpRtcpFdxCtaHd;

       if( (ret = mspSendCommand(*MspOutHd, mspBuildCommand(MSP_ENDPOINT_RTPFDX,
MSP_CMD_RTPFDX_CHG_QUOTA),
    pRtpAssemblerBuffer,sizeof(msp_ENDPOINT_RTPFDX_CMD) )))
        {
          printf("mspSendCommand() api failed \n");
        }
        ret = WaitForSpecificEvent(*CtaOutHd,
    (MSPEVN_SENDCOMMAND_DONE|MSP_CMD_RTPFDX_CHG_QUOTA), &event );

        if ( ret == FAILURE)
        {
          printf("Mspp Send Command failed \n");
        }
        else if ( event.value != CTA_REASON_FINISHED )
        {
          printf("Mspp Send Command failed \n");
          ret = mspReleaseBuffer(event.objHd, event.buffer);
          if ( ret != SUCCESS)
            printf("Mspp release buffer failed...return code = %x\n",ret);
        }
        else
        {
          printf("Frames Per Packet changed to %d\n",nFramesPerPkt);
          ret = mspReleaseBuffer(event.objHd, event.buffer);
          if ( ret != SUCCESS)
            printf("Mspp release buffer failed...return code = %x\n",ret);
        }
```

The application uses the **mspBuildCommand** macro to build the filter command, and the **mspSendCommand** function to send this filter-specific command to the DS0 endpoint.

Sending commands to channel filters

Once an application creates an MSPP channel and connects it to a pair of endpoints, the application can control the component channel filters with **mspSendCommand**. **mspSendCommand** conveys filter-specific commands to filters associated with connected channels.

Channel commands are executed by the filters that make up the channel. Each filter that makes up the channel responds to a specific set of commands. Command information is passed through a predefined structure.

To send a command to a filter within a connected channel, the application invokes **mspSendCommand** and specifies the following:

- MSPP ***chnhd***.
- Filter command produced by using the OR operator to combine a filter ID with a filter command ID.

  To build a valid filter command, use the macro **mspBuildCommand** (defined in *mspcmd.h)* to concatenate the channel filter ID with an endpoint command ID.
- Pointer to a filter-specific command structure.
- Size of the specified structure.

Voice encoder filters, voice decoder filters, and T.38 fax relay filters receive commands only when the parent channel is connected and enabled. Applications must wait for a done event (MSPEVN_SENDCOMMAND_DONE) before sending additional filter commands to any MSPP filter. If the channel contains multiple filters, the application can send simultaneous commands to all of the channel filters and wait for the separate events.

MSPP channel filters include:

Filter ID	Description
MSP_FILTER_G711_Encoder	Encodes data to G.711 64 kbit/s mu-law or A-law form without VAD.
MSP_FILTER_G711_Decoder	Decodes data from G.711 64 kbit/s mu-law or A-law form.
MSP_FILTER_G723_Encoder	Encodes data to G.723.1/A 6.4/5.3 kbit/s form with VAD.
MSP_FILTER_G723_Decoder	Decodes data from G.723.1/A 6.4/5.3 kbit/s form.
MSP_FILTER_G726_Encoder	Encodes data to G.726 32 kbit/s form without VAD.
MSP_FILTER_G726_Decoder	Decodes data from G.726 32 kbit/s form.
MSP_FILTER_G729A_Encoder	Encodes data to G.729A/B 8.0 kbit/s form with VAD.
MSP_FILTER_G729A_Decoder	Decodes data from G.729A/B 8.0 kbit/s form.
MSP_FILTER_JITTER	Receives asynchronous packets and processes them to produce an ordered synchronous output.
MSP_FILTER_T38FAX_FULLDUPLEX	Demodulates the signal received from a PSTN line, stores the data in a fax media packet, and modulates the appropriate signal to be sent over the PSTN.
MSP_FILTER_RTP_SWITCH	Transfers a simplex voice data stream between two RTP endpoints.

For a list of the commands available for each type of MSPP channel filter, refer to Channel filter summary or to the *mspcmd.h* file.

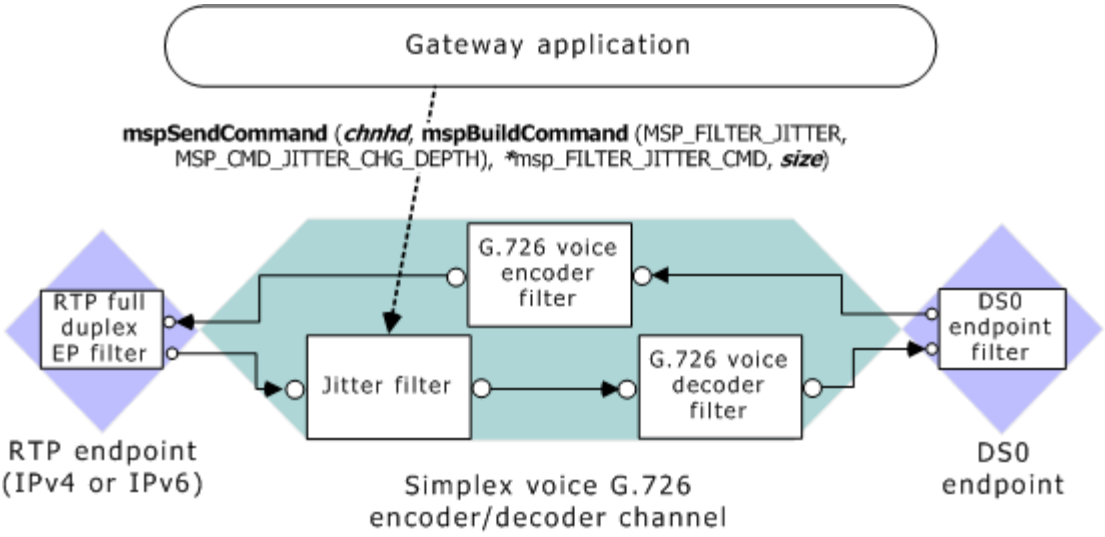
Example 1: Controlling jitter on a G.726 channel

An application changes a jitter filter's jitter depth by changing the number of packets the filter holds in a queue to verify the sequence of the packets. For information about the commands that apply to the jitter filter, refer to the Channel filter summary.

The following illustration shows how an application can control jitter depth by sending a command to a channel's jitter filter with **mspSendCommand** (after the channel is connected). The application provides the ***chnhd*** associated with the G.726 channel, specifies the filter ID and filter command ID (combined using the OR operator), and provides a pointer to the msp_FILTER_JITTER_CMD structure shown here.

```
typedef struct tag_msp_FILTER_JITTER_CMD {
DWORD value;
} msp_FILTER_JITTER_CMD;
```

The value field in the msp_FILTER_JITTER_CMD structure specifies the value to be set for the jitter. The application uses the **mspBuildCommand** macro to concatenate the filter ID and the filter command ID. The following illustration shows and example of sending an MSPP channel command:



The MSPP API returns MSPEVN_SENDCOMMAND_DONE after the command is executed.

The following code sample shows how to use the **mspBuildCommand** macro with **mspSendCommand** to change the jitter depth setting in a MSPP voice channel:

```
msp_FILTER_JITTER_CMD jitter_cmd;
jitter_cmd.value = 2;
jitter_cmd.value = H2NMS_DWORD(jitter_cmd.value); // OS independent endian adjustment

cmd = mspBuildCommand(MSP_FILTER_JITTER, MSP_CMD_JITTER_CHG_DEPTH); // macro
ret = mspSendCommand( hObject, cmd, (void*)&jitter_cmd, sizeof(jitter_cmd));

if (ret != SUCCESS)
    return FAILURE;

// Wait for done event
memset (&Event, 0, sizeof(Event));
ctaWaitEvent( hCta, &Event, CTA_WAIT_FOREVER );

expected_id = MSPEVN_SENDCOMMAND_DONE | MSP_CMD_JITTER_CHG_DEPTH;

// Check for a valid event
if ( Event.id != expected_id ||
     Event.value != CTA_REASON_FINISHED )
{
// Jitter depth change failed (or out of sequence event)
}
else
{
    // Jitter depth change successful
}

// Must release the buffer
if(Event.id == expected_id && Event.size != 0 && Event.buffer != NULL  )
{
    ret = mspReleaseBuffer( Event.objHd, Event.buffer);
    Event.buffer = NULL;
    Event.size = 0;
    if ( ret != SUCCESS)
        return FAILURE;
}
```

For information about MSPP API DONE events, refer to MSPP events.

Example 2: Setting gain for a G.723.1 encoder filter

The following example shows how an application uses **mspSendCommand** to set the encoder gain to 3 dB in a G.723.1 duplex voice channel (the channel has been connected and enabled).

When using **mspSendCommand** to set gain for an MSPP voice encoder (or decoder) filter, the application must define the following values in the **mspSendCommand** buffer:

Value	Example
type	FILTER
objectname	G723ENCODE
description	GAINVALUE

These values are defined according to the following format:

```
msp_[type]_[objectname]_[description]
```

The application then uses the **mspBuildCommand** macro to concatenate the filter ID (for example, MSP_FILTER_G723_ENCODER) and the filter command ID (MSP_CMD_ENCODE_GAIN) when invoking **mspSendCommand**.

The following sample code shows how to set the gain value in an MSPP G.723 encoder channel filter:

```
/* Increase G.723 encoder filter gain by 3 dB                    */
msp_FILTER_ENCODE_CMD EncodeBuffer;

EncodeBuffer.value = 0x05A0; // 3 dB gain from the table shown in
Calculating vocoder filter gain and loss
EncodeBuffer.value = H2NMS_WORD(EncodeBuffer.value); // OS independent endian adjustment

cmd = mspBuildCommand(MSP_FILTER_G723_ENCODER, MSP_CMD_ENCODE_GAIN);
// macro
ret = mspSendCommand( hObject, cmd, (void*)&EncodeBuffer, sizeof(EncodeBuffer));

if (ret != SUCCESS)
    return FAILURE;

// Wait for done event
memset (&Event, 0, sizeof(Event));
ctaWaitEvent( hCta, &Event, CTA_WAIT_FOREVER );

expected_id = MSPEVN_SENDCOMMAND_DONE | MSP_CMD_ENCODE_GAIN;

// Check for a valid event
if ( Event.id != expected_id ||
     Event.value != CTA_REASON_FINISHED )
{
    // Filter gain change unsuccessful
}
else
{
    // Filter gain change successful
}

// Must release the buffer
if(Event.id == expected_id && Event.size != 0 && Event.buffer != NULL  )
{
    ret = mspReleaseBuffer( Event.objHd, Event.buffer);
    Event.buffer = NULL;
    Event.size = 0;
    if ( ret != SUCCESS)
        return FAILURE;
}
```

Setting multiple parameters with a single command

Most MSPP endpoint and channel filters support a command for changing any or all parameters associated with that filter. When using these commands, the application provides a pointer to a structure that specifies appropriate parameters for the filter. For example, an application can use the MSP_CMD_ENCODE_ALL command to change multiple parameters associated with a voice encoder filter by specifying parameters values in a msp_FILTER_ENCODE_ALL_CMD structure.

The following table shows commands available for changing multiple parameters for MSPP endpoint filters:

Filter	Command	Associated structure
RTP full duplex IPv4	MSP_CMD_RTPFDX_CONFIG	msp_ENDPOINT_RTPFDX_CONFIG

Filter	Command	Associated structure
RTP simplex receive IPv4	MSP_CMD_RTPIN_CONFIG	msp_ENDPOINT_RTPFDX_CONFIG
RTP simplex send IPv4	MSP_CMD_RTPOUT_CONFIG	msp_ENDPOINT_RTPFDX_CONFIG
RTP full duplex IPv6	MSP_CMD_RTPFDX_V6_CONFIG	msp_ENDPOINT_RTPFDX_V6_CONFIG
RTP simplex receive IPv6	MSP_CMD_RTPIN_V6_CONFIG	msp_ENDPOINT_RTPFDX_V6_CONFIG
RTP simplex send IPv6	MSP_CMD_RTPOUT_V6_CONFIG	msp_ENDPOINT_RTPFDX_V6_CONFIG
T38UDP	MSP_CMD_T38UDP_CONFIG	msp_ENDPOINT_T38UDP_CONFIG
TPKT	MSP_CMD_TPKT_CONFIG	TPKT_ENDPOINT_PARMS

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

The following table lists commands available for changing multiple parameters for MSPP channel filters:

Filter	Command	Associated structure
Jitter	MSP_CMD_JITTER_INIT	msp_FILTER_JITTER_CMD_INIT
T.38 fax	MSP_CMD_FAXRELAY_CONFIG	MSP_FAX_CHANNEL_PARMS
Voice decoder	MSP_CMD_DECODE_ALL	msp_FILTER_DECODE_ALL_CMD
Voice encoder	MSP_CMD_ENCODE_ALL	msp_FILTER_ENCODE_ALL_CMD

When using these commands, an application can change specific parameters without changing others by specifying 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters for parameters that stay the same. The application specifies new values for the parameters it wants to change.

For example, an application can change the gain level and enable VAD of a voice encoder filter by using the MSP_CMD_ENCODE_ALL filter command. The application can set new values for Gain and VadControl parameters and leave the remaining parameters unchanged by specifying 0xFFFF (or 0xFFFFFFFF).

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Converting command and query structure byte order

When the CG board receives data, it expects the byte ordering in little endian format, where bytes with the higher address (those furthest to the right) are most significant. However, certain operating systems (such as SPARC Solaris) use big endian byte ordering and consider the lower address bytes (bytes furthest to the left) most significant.

Use the following macros to convert values from host byte ordering to board byte ordering format for operating systems that use big endian byte order:

```
H2NMS_DWORD(DWORD val)
H2NMS_WORD(WORD val)
```

To convert values from CG board byte ordering to big endian byte ordering format, use the following macros:

```
NMS2H_DWORD(DWORD val)
NMS2H_WORD(WORD val)
```

Using these macros does not harm systems that use little endian byte ordering. Because of this, we recommend that you use them whenever dealing with command or query structures.

For host systems that use big endian byte ordering, applications must convert the following parameter structures from big endian to little endian (or vice versa) format:

- Structures created by the application and sent to the board with **mspSendCommand**.
- Structures expected from the board in response to **mspSendQuery** calls.

Not all fields within these structures are effected. Only unsigned short (defined as WORD in *nmstypes.h*) and unsigned long (defined as DWORD in *nmstypes.h*) data types are effected.

Example

The following example code shows how to use the byte order conversion macros:

```
DWORD               dword;
WORD                word;

/* From host to board byte ordering format */
   dword  = H2NMS_DWORD(dword);
   word = H2NMS_WORD(word);

// From board to host byte ordering format */
   dword = NMS2H_DWORD(dword);
   word = NMS2H_WORD(word);
```

These macros do not need to be wrapped around an #ifdef _OS_ preprocessor. On systems that do not need any conversion, output from these macros is the same as input.

Sending queries to MSPP filters

The MSPP service provides query commands to solicit configuration and operation data from MSPP filters. Queries (and commands) to endpoint and channel filters return filter status information in a buffer. The buffer must be released using **mspReleaseBuffer**. Applications can query endpoint filters at any time, but can only query channel filters when the channel is enabled.

To send a query to an endpoint or channel filter within a channel, the application can invoke **mspSendQuery** and specify:

- MSPP ***ephd*** or ***chnhd***.
- Filter query produced by ORing a filter ID with a filter query ID.

To build a valid filter query, use the **mspBuildQuery** macro (defined in *mspquery.h)* to concatenate the filter ID with a filter query ID. **mspBuildQuery** is defined as follows:

```
/* Macro used to concatenate queries with filter definitions */
#define mspBuildQuery(filterid,queryid)   ((filterid << 8) | query)
```

The MSPP service returns a filter-specific structure that provides information about the filter's configuration and status.

Note: In operating systems that use big endian byte order (for example, SPARC Solaris), the parameter structures must be converted from big endian to little endian (the byte order format used by the CG board) or vice versa. For information about macros provided to convert from big endian formatted structures to little endian format (and the reverse), refer to Converting command and query structure byte order.

Example: Using mspSendQuery

The following code sample shows how to use **mspSendQuery** to query information about a jitter filter within a voice channel:

```
ret = mspSendQuery(chnhd,
       mspBuildQuery( MSP_FILTER_JITTER, MSP_QRY_JITTER_GET_STATE ),

/* Call a routine to wait for this specific event */
WaitForSpecificEvent( hMsppVoiceChanCtaHd, ( MSPEVN_QUERY_DONE |
MSP_QRY_JITTER_GET_STATE ), &event );

if ( event.value != CTA_REASON_FINISHED )
{
       printf("MSPP Send Command failed \n");
       if ( event.value == CTAERR_OUT_OF_RESOURCES)
       printf("Ran out of buffers probably forgot to release buffers back
       to the service \n");
}
else
{
  if ( event.size != 0 && event.buffer!= 0 )
 /* Since = CTA_REASON_FINISHED should have a buffer, this code is     */
 /* just double checking that a buffer was returned.                   */
    {/* Print out the parts of the structure that we are interested in  */
    pJitterStateBuf = (msp_FILTER_JITTER_STATE *)event.buffer;
    printf("rx = %d\n",pJitterStateBuf->rx);
    printf("rx_accept = %d\n",pJitterStateBuf->rx_accept);
    printf("tx_valid = %d\n",pJitterStateBuf->tx_valid);
    printf("Query Successful\n");
        /* Now that we have data, free buffer back to the service       */
        ret  = mspReleaseBuffer( hMsppVoiceChanCtaHd, event.buffer);
        if ( ret != SUCCESS)
        printf("Mspp release buffer failed...probably CTAERR_NOT_FOUND the
        wrong address sent or it was released already \n");
    }
}
```

The application frees the event buffer by calling **mspReleaseBuffer** and specifying the buffer address.

When applications invoke **mspSendQuery**, the MSPP service returns MSPEVN_QUERY_DONE. The lower byte of event.ID indicates the query ID, and the event.objHD indicates the endpoint or channel filter handle of the MSPP filter that performed the command. For example, if the application uses **mspSendCommand** to set jitter depth, then the MSPEVN_QUERY_DONE ID is 0x001B2201 (the event ID for MSPEVN_QUERY_DONE is 0x001B22***xx*** and the command ID for MSP_CMD_JITTER_CHG_DEPTH is 01).

Processing command and query events

Filter command events provide specific information about endpoint and channel filter events:

Command type	Event information
endpoint filter	The lower byte of the event.ID field in MSPEVN_SENDCOMMAND_DONE identifies the command. The event.obj field specifies the associated endpoint handle
channel filter	The lower byte of the event.ID field in MSPEVN_SENDCOMMAND_DONE identifies the command ID. The event.objHd field specifies the associated channel handle. In addition, the event.buffer field provides a pointer to the filter ID (since there may be more than filter in the channel) and may include additional return data.

The following table shows information included in MSPEVN_SENDCOMMAND_DONE returned in response to **mspSendCommand**:

This field...	Specifies...
event.id	Command ID of the command.
event.objHd	***ephd*** or ***chnhd*** of the endpoint or channel to which the command was directed.
event.value	CTA_REASON_FINISHED or an error.
event.buffer	For channel filter commands, specifies the channel filter ID (depending on the value specified in the event.size field, this field may also return additional information).
event.size	When set to 0, the buffer contains only a filter ID. When not set to 0, the buffer contains additional data returned by the MSPP API (for example RTCP information).

The lower byte of the MSPEVN_SENDCOMMAND_DONE event.id field specifies the command ID of the associated filter command. The event.obj field specifies the endpoint or channel handle of the MSPP component that received the command. For example, if an application uses **mspSendCommand** to set jitter depth, then the MSPEVN_SENDCOMMAND_DONE event.id value is:

```
0x001B2100
```

The event.id field is 0x001B21***xx*** and the command ID for MSP_CMD_JITTER_CHG_DEPTH is 00. The event.value field for the event indicates the status of the command (SUCCESS or a failure reason code).

Because each MSPP filter can support up to 240 unique commands, the range of command codes for each filter is between 0x00 - 0xEF. After receiving MSPEVN_SENDCOMMAND_DONE, the application must free the associated buffer with **mspReleaseBuffer**.

5. Function summary

Endpoint functions

MSPP endpoints provide a point of origin for streaming media. For example, a 64Kbit/s PCM transport would be associated with a DS0 endpoint. This first endpoint can be connected to a MSPP channel which, which in turn, can be connected to a second RTP endpoint.

Function	Synchronous/ Asynchronous	Description
mspCreateEndpoint	Asynchronous	Creates and configures an MSPP service endpoint. The type of endpoint is specified in the ADDR structure.
mspDestroyEndpoint	Asynchronous	Destroys an endpoint.

Channel functions

MSPP channels provide a way to transform streaming media that flows between two endpoints. Channels are made up of one or more channel filters. Filters are parameter-driven functions that perform a series of specific actions on the media flow. This set of filters is referenced by an alphanumeric channel name.

Function	Synchronous/ Asynchronous	Description
mspCreateChannel	Asynchronous	Creates and configures an MSPP service media channel. The type of channel is specified in the ADDR structure with an alphanumeric value.
mspDestroyChannel	Asynchronous	Destroys a channel.

Connection functions

MSPP connections combine channels and endpoints to form simplex and full duplex data paths. A complete connection requires two endpoints, both of which must be connected to a channel.

Function	Synchronous/ Asynchronous	Description
mspConnect	Asynchronous	Connects an MSPP media channel to two endpoints.
mspDisconnect	Asynchronous	Disconnects an MSPP media channel from two endpoints.

Filter command and query functions

Applications can modify endpoint and channel parameters by sending commands to the filters that make up the endpoint or channel. Applications can also query configuration and status information for particular endpoints and channels by sending queries to the filters that make up the endpoint or channel. In both cases, the MSPP API returns events containing buffers that the application must explicitly release.

Function	Synchronous / Asynchronous	Description
mspSendCommand	Asynchronous	Sends a command to an MSPP endpoint or channel filter.
mspSendQuery	Asynchronous	Provides filter configuration or status information.
mspReleaseBuffer	Synchronous	Frees returned event buffers (generated as a result of **mspSendQuery** or **mspSendCommand** calls) so the application can reuse them. This function executes synchronously.
mspGetFilterHandle	Synchronous	Retrieves a filter ID associated with an MSPP API endpoint filter or channel filter.

Note: **mspSendQuery** and **mspSendCommand** events return buffered information that the application must release with **mspReleaseBuffer**.

Refer to the Endpoint filter summary for a list of supported filters and the commands and queries that apply to them.

6. Function reference

Using the function reference

This section provides an alphabetical reference to the MSPP API functions. A prototype of each function is shown with the function description and details of all its arguments and return values. A typical function includes:

Prototype	The prototype is followed by a listing of the function's arguments. Dialogic data types include: • WORD (16-bit unsigned) • DWORD (32-bit unsigned) • INT16 (16-bit signed) • INT32 (32-bit signed) • BYTE (8-bit unsigned) If a function argument is a data structure, the complete data structure is shown.
Return values	The return value for a function is either SUCCESS or an error code. For asynchronous functions, a return value of SUCCESS indicates the function was initiated; subsequent events indicate the status of the operation. Refer to the Alphabetical error summary for a listing of all errors returned by MSPP functions.
Events	If events are listed, the function is asynchronous and is complete when the DONE event is returned. If there are no events listed, the function is synchronous. Additional information such as reason codes and return values may be provided in the value field of the event. Refer to MSPP events for information about MSPP events and reason codes.
Details	Additional information about the function.
See also	A list of related functions.
Example	Example functions taken from sample application programs shipped with the product. The notation /* ... */ indicates additional code that is not shown.

The MSPP API references the following header files:

Header file	Description
mspcmd.h	MSPP filter commands and IDs for each filter object.

Header file	Description
mspdef.h	Functions, parameter structures, events, and errors within the MSPP API.
mspinit.h	Initialization parameters for MSPP endpoint and channel filters.
mspobj.h	MSPP objects. This file defines unique identifiers that applications may need to use to communicate with an object and identifies the MSPP service objects to which queries and messages are addressed.
mspquery.h	MSPP filter queries for MSPP objects. The file defines the return data structure for each query.
mspunsol.h	MSPP event IDs and definitions for buffers appended in unsolicited events.

mspConnect

Connects a media channel to two endpoints.

Prototype

DWORD **mspConnect** (MSPHD ***ephd1,*** MSPHD ***chnhd,*** MSPHD ***ephd2***)

Argument	Description
ephd1	First MSPP endpoint handle.
chnhd	MSPP channel handle.
ephd2	Second MSPP endpoint handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_INTERNAL_HANDLE	Specified MSPP endpoint or channel is not valid.
MSPERR_INVALID_CONNECTION	The specified MSPP connection cannot be found.

Return value	Description
MSPERR_INVALID_HANDLE	Specified MSPP endpoint or channel is not valid.

Events

Event	Description
MSPEVN_CONNECT_DONE	Generated when the MSPP service connects a media channel to two endpoints in response to an **mspConnect** call. MSPEVN_CONNECT_DONE events return the ***chnhd*** of the connected channel, but not the ***ephd***s associated with the connected endpoints. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_ALLOCATE The MSPP service is unable to allocate on-board resources. MSPRSN_FAILED_TO_INTRACONNECT The MSPP service is unable to connect filters within the channel. MSPRSN_FAILED_TO_INTERCONNECT The MSPP service is unable to connect the channel to an endpoint. The event.objHd indicates the ***chnhd*** of the associated channel.

Details

When invoking **mspConnect**, specify two endpoint handles and a channel handle. These handles specify the endpoints through which the data flows, and the processes that will take place (for example, encoding and decoding) as the transferred data flows through a channel. The order of the endpoints in the argument list must correspond to the channel type.

The following restrictions apply when applications use **mspConnect** to connect pairs of MSPP endpoints to MSPP channels:

Connection type	Channel	Endpoints
Voice	Voice	***ephd1***: RTP (IPv4 or IPv6) ***ephd2***: DS0
RTP switching	Switch	***ephd1***: RTP FDX or RTP receive (IPv4 or IPv6) ***ephd2***: RTP FDX or RTP send (IPv4 or IPv6, but must match ***ephd1***)

Connection type	Channel	Endpoints
T.38 fax	Voice	***ephd1***: T38UDP ***ephd2***: DS0
ThroughPacket	Fax relay	***ephd1***: TPKT ***ephd2***: DS0

MSPEVN_CONNECT_DONE returns the ***chnhd*** of the channel associated with the connection, but does not return the ***ephd***s associated with the connected endpoints. For more information, refer to Connecting MSPP endpoints with channels.

See also

mspCreateChannel, **mspCreateEndpoint**, **mspDisconnect**

Example

```
DWORD ret;
CTAHD                   hCtaHd[MAX_CONNECTIONS];
CTAQUEUEHD              hCtaQueue;
CTA_EVENT               Event;
MSPHD hEp1, hEp2, hChan;
ret = mspConnect(hEp1, hChan, hEp2);
if (ret != SUCCESS)
    return FAILURE;
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );
if ( Event.id    != MSPEVN_CONNECT_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspCreateChannel

Creates an MSPP media channel.

Prototype

DWORD **mspCreateChannel** (CTAHD ***ctahd***, MSP_CHANNEL_ADDR* ***addr,*** MSP_CHANNEL_PARAMETER* ***parm***, MSPHD* ***chnhd***)

Argument	Description
ctahd	Context handle used to open the MSPP service.

addr	Pointer to an MSP_CHANNEL_ADDR structure, as shown:
	`typedef struct tag_MSP_CHANNEL_ADDR`
	`{`
	`  DWORD  size;  /* size of this structure */`
	`  DWORD  nBoard;  /* Channel location (board number) */`
	`  MSP_TYPE_CHANNEL  channelType;  /* Type of channel to create */`
	`  DWORD  FilterAttribs;  /* Enables DSP Filter functions */`
	`} MSP_CHANNEL_ADDR;`

Argument	Description

parm	Pointer to the MSP_CHANNEL_PARAMETER structure, as shown:
	`typedef struct tag_MSP_CHANNEL_PARMS`
	`{`
	`DWORD Size;`
	`MSP_CHANNEL_TYPE channelType;`
	`union`
	`{`
	`MSP_VOICE_CHANNEL_PARMS VoiceParms;`
	`MSP_FAX_CHANNEL_PARMS FaxParms;`
	`MSP_RTP_SWITCHING_CHANNEL_PARMS RtpSwitchingParms;`
	`MSP_UNDEFINED_CHANNEL_PARMS Undefined;`
	`} ChannelParms;`
	`} MSP_CHANNEL_PARAMETER;`
	This structure is not required for RTP switching channels. When creating record channels, applications specify channel parameters in a MSP_VOICE_CHANNEL_PARMS structure. Since record channels only contain a single jitter filter, the application only specifies jitter parameters in the structure.

chnhd	Pointer to an MSPP return channel handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	A function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	The handle is invalid.
CTAERR_INVALID_HANDLE	An invalid handle was passed as an argument to this function.
MSPERR_CHANNEL_NOT_DEFINED	The MSPP channel name specified is not defined.

Events

Event	Description
MSPEVN_CREATE_CHANNEL_DONE	Generated when an MSPP channel is created in response to **mspCreateChannel**. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_ALLOCATE The MSPP service is unable to allocate on-board resources. For example, the necessary DSP resources may not be loaded, or the specified UDP port may not be available. The event.objHd indicates the ***chnhd*** of the created channel.

Details

When invoking **mspCreateChannel**, applications specify a context handle (***ctahd***), channel type, and the board number for the channel. When it creates the MSPP channel, the MSPP service returns a unique channel object handle (***chnhd***) that the application can use to configure and control the channel. Once you have created an MSPP channel and enabled it, you can use it to connect MSPP endpoints to create a media path for voice or fax data. For more information, refer to Creating MSPP channels.

Supported MSPP channels include:

Channel ID	Description
FaxRelayFullDuplex	Processes a full duplex data stream by modulating and demodulating T.38 fax data.
G711DecodeSimplex	Processes a simplex data stream by decoding voice data according to the G.711 algorithm.
G711EncodeSimplex	Processes a simplex data stream by encoding voice data according to the G.711 algorithm.
G711FullDuplex	Processes a full duplex data stream by encoding and decoding voice data according to the G.711 algorithm.
G711RecordChannel	Performs jitter buffering and transfers an incoming data stream from an RTP endpoint to a record filter. G.711 recorder channels contain only a jitter filter and can be used with ADI record functions when there is no need to decode the incoming data stream. G.711 recorder channels do not require DSP resources.
G723DecodeSimplex	Processes a simplex data stream by decoding voice data according to the G.723.1/A algorithm.

Channel ID	Description
G723EncodeSimplex	Processes a simplex data stream by encoding voice data according to the G.723.1/A algorithm.
G723FullDuplex	Processes a full duplex data stream by encoding and decoding voice data according to the G.723.1/A algorithm.
G723RecordChannel	Performs jitter buffering and transfers an incoming data stream from an RTP endpoint to a record filter. G.723.1 recorder channels contain only a jitter filter and can be used with ADI record functions when there is no need to decode the incoming data stream. G.723.1 recorder channels do not require DSP resources.
G726DecodeSimplex	Processes a simplex data stream by decoding voice data according to the G.726 algorithm.
G726EncodeSimplex	Processes a simplex data stream by encoding voice data according to the G.726 algorithm.
G726FullDuplex	Processes a full duplex data stream by encoding and decoding voice data according to the G.726 algorithm.
G726RecordChannel	Performs jitter buffering and transfers an incoming data stream from an RTP endpoint to a record filter. G.726 recorder channels contain only a jitter filter and can be used with ADI record functions when there is no need to decode the incoming data stream. G.726 recorder channels do not require DSP resources.
G729DecodeSimplex	Processes a simplex data stream by decoding voice data according to the G.729A/B algorithm.
G729EncodeSimplex	Processes a simplex data stream by encoding voice data according to the G.729A/B algorithm.
G729FullDuplex	Processes a full duplex data stream by encoding and decoding voice data according to the G.729A/B algorithm.
G729RecordChannel	Performs jitter buffering and transfers an incoming data stream from an RTP endpoint to a record filter. G.729 recorder channels contain only a jitter filter and can be used with ADI record functions when there is no need to decode the incoming data stream. G.729 recorder channels do not require DSP resources.
RTPSwitchingSimplex	Conveys a simplex data stream between two RTP endpoints. The channel does not process the data, but merely passes the data on to the next RTP endpoint. Applications can combine two simplex switch channels to create a duplex data path.

See also

mspDestroyChannel, **mspDisableChannel**, **mspEnableChannel**, **mspSendCommand**

Example

```
mspChanAddr.nBoard = DEFAULT_BOARDID;
mspChanAddr.channelType = G711FullDuplex;

ret = mspCreateChannel( hCta, &mspChanAddr, NULL, &hChan );
if (ret != SUCCESS)
     return FAILURE;
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_CREATE_CHANNEL_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspCreateEndpoint

Creates an MSPP endpoint.

Prototype

DWORD **mspCreateEndpoint** (CTAHD ***ctahd***, MSP_ENDPOINT_ADDR* ***addr,*** MSP_ENDPOINT_PARAMETER* ***parm,*** MSPHD* ***ephd***)

Argument	Description
ctahd	Context handle used to open the MSPP service. You can create only one MSPP endpoint per context.

addr	Pointer to an MSP_ENDPOINT_ADDR structure, as shown:
	`typedef struct tag_MSP_ENDPOINT_ADDR`
	`{`
	`  DWORD Size; // Size of this structure`
	`  DWORD nBoard; // Board number`
	`  DWORD eEpType; // Filter ID for the type of endpoint to create.`
	`  union {`
	`  DS0_ENDPOINT_ADDR  DS0;`
	`  RTPRTCP_ENDPOINT_ADDR  RtpRtcp;`
	`  RTPRTCP_V6_ENDPOINT_ADDR  RtpRtcpV6;`
	`  T38UDP_ENDPOINT_ADDR  T38Udp;`
	`  TPKT_ENDPOINT_ADDR  Tpkt;`
	`  } EP; } MSP_ENDPOINT_ADDR;`
	Note: When the specified context is associated with an ADI service instance, the timeslot specified in the endpoint address structure must be the same timeslot used to open the ADI service on the specified ctahd. For more information, refer to Setting up the NaturalAccess environment.

Argument	Description

parm	Pointer to an MSP_ENDPOINT_PARAMETER structure, as shown:
	
	`typedef struct tag_MSP_ENDPOINT_PARMS`
	`{`
	`  DWORD size; // Size of this structure`
	`  DWORD eParmType; // MSP_ENDPOINT_DS0, MSP_ENDPOINT_RTPFDX, etc`
	`  union {`
	`  DS0_ENDPOINT_PARMS  DS0;`
	`  RTPRTCP_ENDPOINT_PARMS  RtpRtcp;`
	`  RTPRTCP_V6_ENDPOINT_PARMS  RtpRtcpV6;`
	`  T38UDP_ENDPOINT_PARMS  T38Udp;`
	`  TPKT_ENDPOINT_PARMS  Tpkt;`
	`  } EP;`
	`} MSP_ENDPOINT_PARAMETER;`

ephd	Pointer to an MSPP return endpoint handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_DRIVER_COMMAND_FAILED	SPI command failed.

Events

Event	Description
MSPEVN_CREATE_ENDPOINT_DONE	Generated when an MSPP endpoint is created in response to **mspCreateEndpoint**. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_ALLOCATE The MSPP service is unable to allocate on-board resources. The event.objHd indicates the ***ephd*** of the created endpoint.

Details

Applications create MSPP endpoints by invoking **mspCreateEndpoint** and specifying the ***ctahd*** used to open the MSPP service instance. In addition, applications specify parameters in an address structure and a parameter structure that define the endpoint's configuration.

The address structure specifies the type of endpoint to create, and the source and destination addresses for transferring the data. The parameter structure configures specific features that will be available on the MSPP endpoint.

When it creates the MSPP endpoint, the MSPP service returns a unique endpoint object handle (***ephd***) that the application can use to configure and control the endpoint. MSPP endpoints are enabled by default. For more information, refer to Creating MSPP endpoints.

The MSPP service provides the following set of endpoints for sending and receiving data to and from a network interface:

Endpoint type	Filter ID	Provide
DS0 (PSTN)	MSP_ENDPOINT_DS0	Entry or exit points for transferring a voice or T.38 fax data. DS0 endpoints can be connected to simplex or duplex channels.
RTP IPv4 full duplex	MSP_ENDPOINT_RTPFDX	Entry and exit points for full duplex voice over IPv4 streams (includes an RTP header within UDP packets).
RTP IPv4 simplex receive	MSP_ENDPOINT_RTPIN	Entry points for a simplex voice over IPv4 data streams (includes an RTP header within UDP packets).
RTP IPv4 simplex send	MSP_ENDPOINT_RTPOUT	Exit points for simplex voice over IPv4 data streams (includes an RTP header within UDP packets).
RTP IPv6 full duplex	MSP_ENDPOINT_RTPFDX_V6	Entry and exit points for full duplex voice over IPv6 streams (includes an RTP header within UDP packets).
RTP IPv6 simplex receive	MSP_ENDPOINT_RTPIN_V6	Entry and exit points for simplex voice over IPv6 data streams (includes an RTP header within UDP packets).
RTP IPv6 simplex send	MSP_ENDPOINT_RTPOUT_V6	Exit point for simplex voice over IPv6 data streams (includes an RTP header within UDP packets).
TPKT full duplex	MSP_ENDPOINT_TPKT	Entry and exit points for duplex voice data streams in which packets are processes according to the ThroughPacket multiplexing algorithm.

Endpoint type	Filter ID	Provide
T38UDP full duplex	MSP_ENDPOINT_T38FDX	Entry and exit points for full duplex T.38 fax UDP data streams (including those carrying T.38 fax data).

See also

mspDestroyEndpoint, **mspDisableEndpoint**, **mspEnableEndpoint**

Example

```
mspAddr.eEpType = MSP_ENDPOINT_DS0;
mspAddr.nBoard = DEFAULT_BOARDID;
mspAddr.size = sizeof(DS0_ENDPOINT_ADDR);
mspAddr.EP.DS0.nTimeslot = 1;

mspParm.eParmType = MSP_ENDPOINT_DS0;
mspParm.EP.DS0.media = MSP_VOICE;
mspParm.size = sizeof(DS0_ENDPOINT_PARMS);

ret = mspCreateEndpoint( hCta, &mspAddr, &mspParm, &hEp);
if (ret != SUCCESS)
    return FAILURE;

// Wait for event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_CREATE_ENDPOINT_DONE ||
     Event.value != CTA_REASON_FINISHED )
   return FAILURE;
```

mspDestroyChannel

Destroys a specified MSPP channel.

Prototype

DWORD **mspDestroyChannel** (MSPHD ***chnhd***)

Argument	Description
chnhd	MSPP channel handle.

Return values

Return values	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.

Return values	Description
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_INVALID_HANDLE	Specified MSPP channel is not valid.

Events

Event	Description
MSPEVN_DESTROY_CHANNEL_DONE	Generated when an MSPP media channel is destroyed following an **mspDestroyChannel** function call. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_DEALLOCATE MSPP service is unable to de-allocate on-board resources. MSPRSN_FAILED_TO_INTRADISCONNECT MSPP service is unable to disconnect filters within the channel. MSPRSN_FAILED_TO_INTERDISCONNECT MSPP service is unable to disconnect the channel from an endpoint. The event.objHd specifies the ***chnhd*** of the destroyed channel.

Details

When called, **mspDestroyChannel** disconnects and destroys a specified MSPP channel.

See also

mspCreateChannel, **mspDisableChannel**

Example

```
ret = mspDestroyChannel(hChan);
if (ret != SUCCESS)
    return FAILURE;

// Wait for event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_DESTROY_CHANNEL_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspDestroyEndpoint

Destroys a specified MSPP endpoint.

Prototype

DWORD **mspDestroyEndpoint** (MSPHD ***ephd***)

Argument	Description
ephd	MSPP endpoint handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	An invalid handle was passed as an argument to this function.
MSPERR_DRIVER_COMMAND_FAILED	SPI command has failed.
MSPERR_ENDPOINT_NOT_ALLOCATED	MSPP endpoint is defined, but not yet allocated on the board.
MSPERR_INVALID_HANDLE	Specified MSPP endpoint is not valid.

Events

Event	Description
MSPEVN_DESTROY_ENDPOINT_DONE	Generated when an MSPP endpoint is destroyed following a call to **mspDestroyEndpoint**. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_DEALLOCATE MSPP service is unable to deallocate on-board resources. The event.objHd indicates the ***ephd*** of the destroyed endpoint.

Details

When called, **mspDestroyEndpoint** destroys a specified MSPP endpoint, and disables and disconnects the associated connection, if applicable. For more information, refer to Disabling and enabling MSPP endpoints.

See also

mspCreateEndpoint, **mspDisableEndpoint**

Example

```
ret = mspDestroyEndpoint(hEp);
if (ret != SUCCESS)
    return FAILURE;

// Wait for event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_DESTROY_ENDPOINT_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspDisableChannel

Stops the specified MSPP channel from processing data.

Prototype

DWORD **mspDisableChannel** (MSPHD ***chnhd***)

Argument	Description
chnhd	MSPP channel handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_CHANNEL_NOT_CONNECTED	Specified MSPP channel is not connected to any endpoints.
MSPERR_DRIVER_COMMAND_FAILED	SPI command has failed.
MSPERR_INTERNAL_HANDLE	Specified MSPP channel is not valid.

Return value	Description
MSPERR_INVALID_HANDLE	Specified MSPP channel is not valid.

Events

Event	Description
MSPEVN_DISABLE_CHANNEL_DONE	Generated when the flow of data through an MSPP channel is stopped in response to an **mspDisableChannel** function call. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_STOP MSPP service is unable to stop the on-board resource. The event.objHd specifies the ***chnhd*** of the disabled channel.

Details

When called, **mspDisableChannel** discontinues the flow of all data that is sent through the specified MSPP channel.

See also

mspCreateChannel, **mspDestroyChannel**, **mspEnableChannel**

Example

```
ret = mspDisableChannel(hChan);
if (ret != SUCCESS)
    return FAILURE;

// Wait for event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_DISABLE_CHANNEL_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspDisableEndpoint

Stops the flow of data processed through the specified MSPP endpoint.

Prototype

DWORD **mspDisableEndpoint** (MSPHD ***ephd***)

Argument	Description
ephd	MSPP endpoint handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
MSPERR_ENDPOINT_IS_DISABLED	MSPP endpoint is already disabled.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_DRIVER_COMMAND_FAILED	SPI command has failed.
MSPERR_ENDPOINT_BUSY	MSPP command arrived while servicing a previous command.
MSPERR_ENDPOINT_NOT_ALLOCATED	MSPP endpoint is defined, but not yet allocated on the board.
MSPERR_INVALID_HANDLE	Specified MSPP endpoint is not valid.

Events

Event	Description
MSPEVN_DISABLE_ENDPOINT_DONE	Generated when the flow of data through an endpoint is halted in response to an **mspDisableEndpoint** function call. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_STOP MSPP service is unable to stop the on-board resource. The event.objHd indicates the ***ephd*** of the disabled endpoint.

Details

When called, **mspDisableEndpoint** halts the data flow from a specified MSPP endpoint. Only one call to **mspDisableEndpoint** is needed to interrupt the flow of data to multiple channels that are connected to a single endpoint.

Note: To temporarily disable a DS0 endpoint, disable the connected channels by calling **mspDisableChannel**. Do not use **mspDisableEndpoint** to temporarily disable a DS0 endpoint, because data flow is not restored when the endpoint is re-enabled.

See also

mspCreateEndpoint, **mspDestroyEndpoint**, **mspEnableEndpoint**

Example

```
ret = mspDisableEndpoint(hEp);
if (ret != SUCCESS)
    return FAILURE;

// Wait for event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_DISABLE_ENDPOINT_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspDisconnect

Disconnects a media channel between two MSPP endpoints.

Prototype

DWORD **mspDisconnect** (MSPHD ***ephd1***, MSPHD ***chnhd***, MSPHD ***ephd2***)

Argument	Description
ephd1	First MSPP endpoint handle.

Argument	Description
chnhd	MSPP channel handle.
ephd2	Second MSPP endpoint handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_CHANNEL_NOT_CONNECTED	The specified MSPP channel is not connected to any endpoints.
MSPERR_DRIVER_COMMAND_FAILED	An SPI command has failed.
MSPERR_INTERNAL_HANDLE	Specified MSPP endpoint/channel is not valid.
MSPERR_INVALID_CONNECTION	Specified MSPP connection cannot be found.
MSPERR_INVALID_HANDLE	Specified MSPP endpoint/channel is not valid.

Events

Event	Description
MSPEVN_DISCONNECT_DONE	Generated when an MSPP channel is disconnected from two endpoints in response to an **mspDisconnect** function call. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_DEALLOCATE MSPP service is unable to deallocate on-board resources. MSPRSN_FAILED_TO_INTRADISCONNECT MSPP service is unable to disconnect filters within the channel. MSPRSN_FAILED_TO_INTERDISCONNECT MSPP service was unable to disconnect the channel from an endpoint. The event.objHd indicates the ***chnhd*** of the disconnected channel.

Details

When called, **mspDisconnect** disassociates the MSPP endpoints and MSPP channel in a specified connection. The application specifies endpoint handles in the same order it used when creating the connection with **mspConnect**.

For example, in the case of voice channels, ***ephd1*** must point to an RTP endpoint and ***ephd2*** must point to a DS0 endpoint. In the case of fax channels, ***ephd1*** must point to a UDP endpoint (for fax) and ***ephd2*** must point to a DS0 endpoint. The application must disable channels and endpoints associated with the connection (using **mspDisableChannel** and **mspDisableEndpoint**) before using **mspDisconnect**.

See also

mspConnect

Example

```
ret = mspDisconnect( hEp1, hChan, hEp2);
if (ret != SUCCESS)
    return FAILURE;

// Wait for event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_DISCONNECT_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspEnableChannel

Enables the MSPP channel to begin processing data.

Prototype

DWORD **mspEnableChannel** (MSPHD ***chnhd***)

Argument	Description
chnhd	MSPP channel handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_CHANNEL_NOT_CONNECTED	Specified MSPP channel is not connected to any endpoints.
MSPERR_DRIVER_COMMAND_FAILED	SPI command failed.
MSPERR_INTERNAL_HANDLE	Specified MSPP channel is not valid.
MSPERR_INVALID_HANDLE	Specified MSPP channel is not valid.

Events

Event	Description
MSPEVN_ENABLE_CHANNEL_DONE	Generated when the data flow through an MSPP channel is enabled or resumed in response to an **mspEnableChannel** function call. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_START MSPP service is unable to start the on-board resource. The event.objhd indicates the ***chnhd*** of the enabled channel.

Details

Created MSPP channels are disabled, by default, and must be enabled before data is sent or received. When invoking **mspEnableChannel**, specify the channel handle (***chnhd***) of the connected channel. As soon as data is transferred, the connection channel processes the data and passes it on to the endpoint at the opposite end of the connection.

mspEnableChannel can also be used to re-establish data flow through a channel that had previously been disabled.

See also

mspCreateChannel, **mspDisableChannel**

Example

```
ret = mspEnableChannel( hChan);
if (ret != SUCCESS)
    return FAILURE;

// Wait for event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_ENABLE_CHANNEL_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspEnableEndpoint

Starts the flow of data processing through the specified MSPP endpoint.

Prototype

DWORD **mspEnableEndpoint** (MSPHD ***ephd***)

Argument	Description
ephd	MSPP endpoint handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
MSPERR_ENDPOINT_IS_ENABLED	MSPP endpoint is already enabled.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_DRIVER_COMMAND_FAILED	SPI command has failed.
MSPERR_ENDPOINT_BUSY	MSPP command arrived while servicing a previous command.
MSPERR_ENDPOINT_NOT_ALLOCATED	MSPP endpoint is defined, but not yet allocated on the board.
MSPERR_INVALID_HANDLE	Specified MSPP endpoint is not valid.

Events

Event	Description
MSPEVN_ENABLE_ENDPOINT_DONE	Generated when the data flow between MSPP endpoints is halted following a call to **mspEnableEndpoint**. The event.value field can contain the following: CTA_REASON_FINISHED MSPRSN_FAILED_TO_START MSPP service is unable to start the on-board resource. The event.objhd indicates the ***ephd*** of the enabled endpoint.

Details

Created MSPP endpoints are enabled, by default. When invoking **mspEnableEndpoint**, specify the endpoint handle (***ephd***) of the connected endpoint.

mspEnableEndpoint can also be used to re-establish the data flow through a channel that had previously been disabled.

See also

mspCreateEndpoint, **mspDisableEndpoint**

Example

```
ret = mspEnableEndPoint( hEp1);
if (ret != SUCCESS)
    return FAILURE;

// Wait for event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
if ( Event.id    != MSPEVN_ENABLE_ENDPOINT_DONE ||
     Event.value != CTA_REASON_FINISHED )
    return FAILURE;
```

mspGetFilterHandle

Retrieves a filter ID associated with an MSPP API endpoint filter or channel filter.

Prototype

DWORD **mspGetFilterHandle** (MSPHD ***msphd***, DWORD ***filtertype***, DWORD ****fltID***)

Return value	Description
msphd	Endpoint handle (ephd) returned by **mspCreateEndpoint** or channel handle (chnhd) returned by **mspCreateChannel**.
filtertype	MSPP API filter type. Valid filter types include: • MSP_ENDPOINT_RTPOUT • MSP_ENDPOINT_RTPFDX • MSP_ENDPOINT_RTPOUT_V6 • MSP_ENDPOINT_RTPFDX_V6 • MSP_JITTER_FILTER
fltID	Pointer to a filter ID location.

Return values

Event	Description
SUCCESS	
ADIERR_INVALID_CALL_STATE	Function not valid in the current call state. For example, many functions require the call to be in ADI_CC_STATE_CONNECTED.

Event	Description
CTAERR_BAD_ARGUMENT	Specified is ***filtertype*** invalid.
CTAERR_INVALID_CTAHD	Invalid ***msphd*** was specified.
CTAERR_INVALID_STATE	Function is not valid in the current state.

Details

This function retrieves a board-level filter ID (***fltID***) associated with an MSPP channel or endpoint. You can use the ***fltID*** to connect ADI play and record filters to MSPP filters at runtime when implementing Dialogic native play and record.

Retrieving endpoint filter handles

When an application creates an MSPP API endpoint with **mspCreateChannel**, the MSPP API returns an endpoint handle (***ephd***). Applications use **mspGetFilterHandle** to retrieve the ***fltID*** associated with an existing RTP IPv4 or IPv6 endpoint. When invoking **mspGetFilterHandle**, the application specifies the endpoint handle (***ephd***) and the endpoint type (***filtertype***). The application can use the returned ***fltID*** as the ***egresshd*** argument for the ADI API **adiSetNativeInfo** function.

Retrieving channel filter handles

When an application creates an MSPP API channel with **mspCreateChannel**, the MSPP API returns a channel handle (***chnhd***). Applications use **mspGetFilterHandle** to retrieve the jitter filter handle (***fltID***) associated with the jitter filter in an existing MSPP channel. When invoking **mspGetFilterHandle**, the application specifies the channel handle (***chnhd***) and the filter type (in this case MSP_JITTER_FILTER). The application can use the returned fltID as the ***ingresshd*** argument for the ADI API **adiSetNativeInfo** function.

For more information about implementing Dialogic native play and record functionality, refer to the *Dialogic® NaturalAccess™ Fusion VoIP API Developer's Manual*.

Example

```
ret = ctaCreateQueue(NULL, 0, &hCtaQueueHd);
ret = ctaCreateContext(   hCtaQueueHd, 0, "Play", &ctahd );

ServiceCount = 2;
ServDesc[0].name.svcname         = "ADI";
ServDesc[0].name.svcmgrname      = "ADIMGR";
ServDesc[0].mvipaddr.board       = board;
ServDesc[0].mvipaddr.mode        = 0;
ServDesc[1].name.svcname         = "MSP";
ServDesc[1].name.svcmgrname      = "MSPMGR";
ServDesc[1].mvipaddr.board       = board;

ret = ctaOpenServices( ctahd, ServDesc, ServiceCount);
ret = WaitForSpecificEvent( CTAEVN_OPEN_SERVICES_DONE, &event );
ret = adiStartProtocol( ctahd, "nocc", NULL, NULL );
ret = WaitForSpecificEvent( ADIEVN_STARTPROTOCOL_DONE, &event );

// create mspp RTP endpoint
ret = mspCreateEndpoint( ctahd, &mspAddr, &mspParm, &ephd);

// Wait for event
ctaWaitEvent( hCtaQueueHd, &event, CTA_WAIT_FOREVER );

// Check the reason code
if ( event.id    != MSPEVN_CREATE_ENDPOINT_DONE ||
event.value != CTA_REASON_FINISHED )
return FAILURE;

// get cg6xxx board handle
ret = mspGetFilterHandle(msphd, MSP_FILTER_RTPFDX_EPH,
&cg6xxx_board_filter_handle);

// get default adi play parms
ret = adiSetNativeInfo( ctahd,
NULL,  /* no ingress handle, as this is a play only */
cg6xxx_board_filter_handle,
&natpr_ctl); /* cg6xxx board filter handle */

ret = adiPlayFromMemory( ctahd,
ADI_ENCODE_EDTX_AMRNB, /* audio play */
        MemoryBuffer, RecordedBytes, NULL);
```

mspReleaseBuffer

Returns an event buffer to the MSPP API.

Prototype

DWORD **mspReleaseBuffer** (MSPHD ***msphd***, void ****buffer***)

Argument	Description
msphd	MSPP handle associated with the channel or endpoint that sent the data buffer.
buffer	Pointer to the address of the buffer to release.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_DRIVER_COMMAND_FAILED	SPI command has failed.
MSPERR_ENDPOINT_NOT_ALLOCATED	MSPP endpoint is defined, but not yet allocated on the board.
MSPERR_INVALID_HANDLE	Specified MSPP endpoint/channel is not valid.

Events

None.

Details

This function returns retrieved event buffers so the application can reuse them.

The following restrictions apply to **mspReleaseBuffer**:

- If an application uses an endpoint or a channel with an attached buffer but does not return the buffer (through **mspReleaseBuffer**), the number of events the applications receives is limited to ten. Subsequent events are dropped.
- Calling **mspReleaseBuffer** twice for the same buffer creates an access violation.

See also

mspSendQuery, **mspSendCommand**

Example

```
if(Event.size != 0 && Event.buffer != NULL  )
{
    ret = mspReleaseBuffer( Event.objHd, Event.buffer);
    Event.buffer = NULL;
    Event.size = 0;

    if ( ret != SUCCESS)
        return FAILURE;

}
```

mspSendCommand

Sends a command to a MSPP object for processing.

Prototype

DWORD **mspSendCommand** (MSPHD ***msphd***, DWORD ***command***, void ****buffer***, DWORD ***size***)

Argument	Description
msphd	MSPP endpoint or channel handle associated with the filter to which the command is directed.
command	Macro used to concatenate commands with filter definitions.
buffer	Pointer to a buffer that holds the values or parameters of the command.
size	Size, in bytes, of buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_DRIVER_COMMAND_FAILED	SPI command has failed.
MSPERR_ENDPOINT_NOT_ALLOCATED	MSPP endpoint is defined but not yet allocated on the board.
MSPERR_ENDPOINT_BUSY	MSPP command arrived while servicing a previous command or query.
MSPERR_FILTER_BUSY	MSPP command arrived while servicing a previous command/query.
MSPERR_INTERNAL_HANDLE	Specified MSPP endpoint/channel is not valid.

Return value	Description
MSPERR_INVALID_HANDLE	Specified MSPP endpoint/channel is not valid.
MSPERR_SENDCOMMAND_FAILED	**mspSendCommand** returned an error.
MSPERR_UNKNOWN_FILTER_OBJECT	Command specified a filter than is not in the channel.

Events

Event	Description
MSPEVN_SENDCOMMAND_DONE	Generated when a command is successfully sent to an MSPP object for processing. The last two digits of this event indicate the command response. The event.objhd field provides the object handle of the endpoint or channel that is responding, and an included buffer contains a pointer to the filter ID. For more information, refer to the Details section.

Details

This function sends a command to a specified MSPP endpoint or channel filter. After an application creates an MSPP channel or connects and enables an MSPP channel, it can re-configure the associated filters with **mspSendCommand**. All endpoints except DS0 endpoints must be in a disabled state before they can receive commands (DS0 endpoints can receive endpoints while enabled). All channel filters except the jitter filter can receive commands after the channel is connected and enabled (jitter filters can receive commands as soon as the channel is created). Applications must use **mspReleaseBuffer** to release any buffers returned MSPEVN_SENDCOMMAND_DONE.

When invoking **mspSendCommand**, the application specifies a channel or endpoint handle, a valid filter ID concatenated with a valid command ID, and a buffer where you have specified command information.

Channel commands are executed by the filters that make up the channel. Each filter that makes up the channel responds to a specific set of commands, and command information is passed in the form of a predefined structure. For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Each endpoint and channel filter responds to one command that can be used to configure all the parameters associated with that filter. When sending these commands, applications can specify new values for the parameters they want to change, and set the values to 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) for the parameters for which they want to keep the existing settings. For more information about the configuration commands that apply to different MSPP filters, refer to the specific channel or endpoint filter, or to Setting multiple parameters with a single command.

The lower byte of the event.ID field within MSPEVN_SENDCOMMAND_DONE provides the command ID of the filter command. The event.objHd specifies the MSPP object handle of the endpoint or channel filter that receives the command. Since MSPP endpoints are associated only with a single endpoint filter, this information is enough to determine both the success or the failure of the endpoint command, and the identity of the endpoint that received the command.

Because some MSPP channels consist of multiple MSPP channel filters (for example, a duplex voice channel includes a voice decoder, a voice encoder, and a jitter filter), MSPEVN_SENDCOMMAND_DONE returns additional information to distinguish the specific channel filter that received the command. For channel filter commands, the event.buffer field within the DONE event provides a pointer to the specific channel filter ID to which the command was directed.

The following table lists the information contained within **mspSendCommand** DONE events:

Field	Description
event.objHd	***ephd*** or ***chnhd*** of the object to which the command is directed.
event.buffer, event.size	Always contains a pointer to the filter ID, and can contain a pointer to additional data.

Applications can send commands and queries to any MSPP service endpoint or channel filter. However, each filter type can receive commands and events only in particular states. For information about the states in which MSPP endpoint and channel filters can receive filter commands, refer to When to send filter commands and queries.

See also

mspConnect, **mspCreateChannel**, **mspCreateEndpoint**, **mspSendQuery**

Example

```
msp_FILTER_JITTER_CMD jitter_cmd;
jitter_cmd.value = 2;
jitter_cmd.value = H2NMS_DWORD(jitter_cmd.value); // OS independent endian adjustment

DWORD command = mspBuildCommand(MSP_FILTER_JITTER, MSP_CMD_JITTER_CHG_DEPTH);

ret = mspSendCommand( hObject, command, (void*)&jitter_cmd, sizeof(jitter_cmd);

if (ret != SUCCESS)
    return FAILURE;

// Wait for done event
ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
DWORD expected_id = MSPEVN_SENDCOMMAND_DONE | MSP_CMD_JITTER_CHG_DEPTH;

if ( Event.id != expected_id ||
     Event.value != CTA_REASON_FINISHED )
{

// Jitter depth change failed (or out of sequence event)
}
else
{
    // Jitter depth change successful
}
```

```
// Release the command buffer in all cases
if(Event.size != 0 && Event.buffer != NULL  )
{
    ret = mspReleaseBuffer( Event.objHd, Event.buffer);
    Event.buffer = NULL;
    Event.size = 0;
    if ( ret != SUCCESS)
        return FAILURE;
}
```

mspSendQuery

Provides MSPP filter configuration or status information.

Prototype

DWORD **mspSendQuery** (MSPHD ***msphd***, DWORD ***query***)

Argument	Description
msphd	MSPP endpoint or channel handle associated with the filter to which the query is directed.
query	Valid filter query used to connect commands with filter definitions.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument includes an invalid value or a required pointer argument is NULL.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_HANDLE	Invalid handle was passed as an argument to this function.
MSPERR_DRIVER_COMMAND_FAILED	SPI command has failed.
MSPERR_ENDPOINT_BUSY	MSPP query arrived while servicing a previous command/query.
MSPERR_ENDPOINT_NOT_ALLOCATED	MSPP endpoint is defined, but not yet allocated on the board.
MSPERR_FILTER_BUSY	MSPP query arrived while servicing a previous command/query.
MSPERR_INTERNAL_HANDLE	Specified MSPP endpoint/channel is not valid.

Return value	Description
MSPERR_INVALID_HANDLE	Specified MSPP endpoint/channel is not valid.
MSPERR_SENDQUERY_FAILED	MSPP query could not be sent.
MSPERR_UNKNOWN_FILTER_OBJECT	Command specified a filter than is not in the channel.

Events

Event	Description
MSPEVN_QUERY_DONE	Generated when a query is successfully sent to an MSPP object for processing. The last two digits of this event indicate the command response. The event.objhd field provides the object handle of the endpoint or channel that is responding, and an included buffer contains a pointer to the filter ID. For more information, refer to Details.

Details

After an application creates an MSPP channel or connects and enables an MSPP channel, it can use **mspSendQuery** to solicit configuration and operation data from the associated MSPP filters.

When invoking **mspSendQuery**, specify a channel or endpoint handle, a valid filter ID concatenated with a valid query ID, a buffer where the buffered query data can be returned, and the size of the query structure to be returned. When an application uses **mspSendQuery** to send a system-level query (to query CPU utilization or route availability information), the application must specify the constant MSP_SYSTEM as the filter ID.

mspSendQuery sends a query to a specified MSPP endpoint or channel filter, which returns a filter-specific structure that provides filter-specific configuration and status information.

The following structure is an example of return data for an **mspSendQuery** call for an RTP endpoint filter:

```
typedef struct tag_msp_FILTER_RTPDISASM_STATE
{
DWORD   filter_id;
DWORD   rx;
DWORD   rx_accept;
DWORD   tx;
DWORD   tx_valid;
DWORD   last_pid_rcvd;
DWORD   mismatches;
DWORD   last_map_vocoder;
DWORD   last_map_pid;
DWORD   dtmf_frames;
DWORD   dtmf_event_ctrl;

} msp_FILTER_RTPDISASM_STATE;
```

The lower byte of the event.ID field within MSPEVN_SENDQUERY_DONE provides the query ID of the filter query. The event.objHd specifies the MSPP handle with which the endpoint or channel filter is associated. Since each MSPP endpoint is associated with a single endpoint filter, this information is enough to determine the success or failure of the query and the identity of the endpoint that received the command.

Because some MSPP channels consist of multiple MSPP filters (for example, a duplex voice channel includes a voice decoder filter, a voice encoder filter, and a jitter filter), MSPEVN_SENDQUERY_DONE returns additional information to distinguish the specific channel filter that received the query. For channel filter queries, the event.buffer field within the DONE event provides a pointer to the specific channel filter ID to which the query was directed.

The following table shows information the application can obtain from the **mspSendQuery** DONE event:

Field	Description
event.objHd	***ephd*** or ***chnhd*** of the object to which the query is directed.
event.buffer, event.size	Pointer to the filter ID, and optionally a pointer to additional data. The application must return the buffer with **mspReleaseBuffer**.

Applications can send commands and queries to any MSPP service endpoint or channel filter. However, each filters type can receive commands and events only in particular states. For information about the states in which specific MSPP endpoint and channel filters can receive filter queries, refer to When to send filter commands and queries. For a list of endpoint and channel filters and associated queries, refer to the specific channel or endpoint filter, or to Setting multiple parameters with a single command.

See also

mspReleaseBuffer, **mspSendCommand**

Example

```
DWORD query = mspBuildQuery(MSP_FILTER_JITTER,MSP_QRY_JITTER_GET_STATE);

ret = mspSendQuery(hObject, query);
if (ret != SUCCESS)
    return FAILURE;

ctaWaitEvent( hCtaQueHd, &Event, CTA_WAIT_FOREVER );

// Check the reason code
DWORD expected_id = MSPEVN_SENDQUERY_DONE | MSP_CMD_JITTER_CHG_DEPTH;

if ( Event.value != CTA_REASON_FINISHED !!
     Event.id != expected_id)
{
    // Query failed
}
else
{
    if ( Event.size > sizeof(msp_FILTER_JITTER_STATE) &&
         Event.buffer!= 0 )
    {
        // Query Successful - grab the data
    }
}

// Release the buffer in all cases
if(Event.size != 0 && Event.buffer != NULL  )
{
    ret = mspReleaseBuffer( Event.objHd, Event.buffer);
    if ( ret != SUCCESS)
        return FAILURE;
}
```

7. Endpoint filter reference

Endpoint filter summary

Each MSPP endpoint consists of a single endpoint filter. Endpoint filters provide an entry or exit point (or both) through which data can flow. MSPP endpoint filters include:

Endpoint filter	Description
DS0 full duplex	Provides an entry or exit point for data received from or sent to a PSTN interface. This filter can send and receive simplex or duplex data streams.
RTP IPv4 full duplex	Provides an entry or exit point for duplex packetized voice data streams received from or sent to an IPv4 interface, and performs the following processing tasks: • Creates compound packets that contain one or more frames of encoded media. • Disassembles compound packets into separate unpacked frames.
RTP IPv4 simplex receive	Provides an entry point for simplex packetized voice data streams received from an IPv4 interface and disassembles compound packets to separate unpacked frames.
RTP IPv4 simplex send	Provides an exit point for simplex packetized voice data streams sent to an IPv4 interface. Also creates compound packets that contain one or more frames of encoded media.
RTP IPv6 full duplex	Provides an entry or exit point for duplex packetized voice data streams received from or sent to an IPv6 interface, and performs the following processing tasks: • Creates compound packets that contain one or more frames of encoded media. • Disassembles compound packets into separate unpacked frames.
RTP IPv6 simplex receive	Provides an entry point for simplex packetized voice data streams received from an IPv6 interface and also disassembles compound packets to separate unpacked frames.
RTP IPv6 simplex send	Provides an exit point for simplex packetized voice data streams sent to an IPv6 interface. Also creates compound packets that contain one or more frames of encoded media.
T38UDP	Converts T.38 UDP packets into fax data and vice versa (these endpoints do not support IPv6).

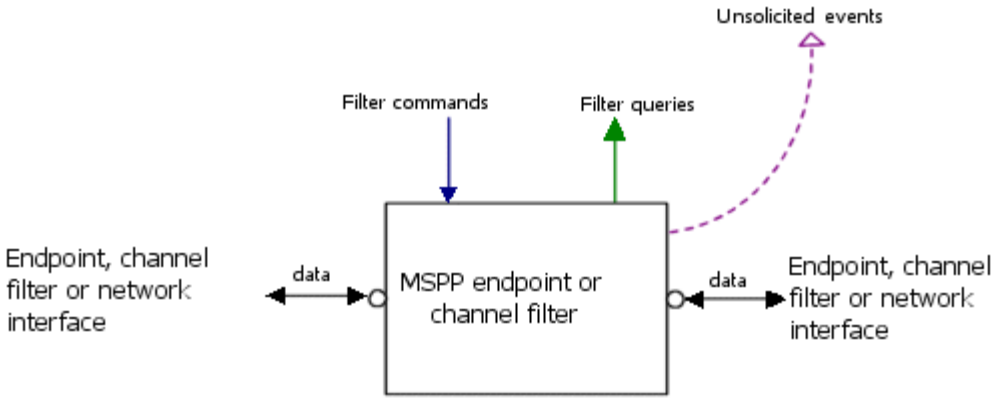
Endpoint filter	Description
TPKT	Provides an entry and exit point for full duplex voice data stream multiplexed according to the ThroughPacket algorithm (these endpoints do not support IPv6).

Using the endpoint filter reference

This section provides a comprehensive, alphabetically-ordered reference to Fusion MSPP filters. A typical filter description includes:

Filter ID	ID for the filter as specified in the *mspcmd.h* file.
Initialization structure	Structure used to specify filter parameters at initialization.
Restrictions	Types of standard MSPP connections that use the filter.
Filter connections	List of MSPP filters from which the filter can send and receive data.
Commands	List of valid commands that can be sent to the filter with **mspSendCommand**.
Queries	List of valid queries that can be sent to the filter with **mspSendQuery.**
Unsolicited events	Filter-specific events returned to the application through the MSPP service.
Details	Detailed information about the MSPP filter and its configuration.
See also	References to related filters.

Each filter description illustrates the filter in the following way:



DS0 full duplex endpoint filter

Provides an entry and exit point for duplex voice and fax data at the PSTN interface. For information about DS0 endpoint filter parameters, refer to DS0 endpoint address structure and DS0 endpoint parameter structure, and to the *mspdef.h* header file.

Filter ID

MSP_ENDPOINT_DS0

Initialization structure

DS0_ENDPOINT_PARMS

Restrictions

Used for duplex and simplex voice connections and T.38 fax connections.

Channel connections

Allowed connections include:

- Full duplex voice channels
- Simplex voice channels
- Full duplex T.38 fax channels

Commands

DS0 full duplex endpoint filters respond to the following command any time after the endpoint is created:

Command ID	Description
MSP_CMD_DS0_CONFIG	Sends a DS0_ENDPOINT_PARMS structure for a particular endpoint filter. The structure specifies whether the endpoint transmits data in voice or fax transmission mode (set to voice by default).

When sending MSP_CMD_DS0_CONFIG commands to DS0 endpoint filters, include a pointer to the following structure:

```
typedef struct tag_DS0_ENDPOINT_PARMS{
DWORD size
   MSP_MEDIA media;
} DS0_ENDPOINT_PARMS;
```

The MSP_MEDIA parameter is defined as an enumerated data type where MSP_VOICE is the first defined (0) value, and MSP_FAX is the second (1) value.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Queries

None.

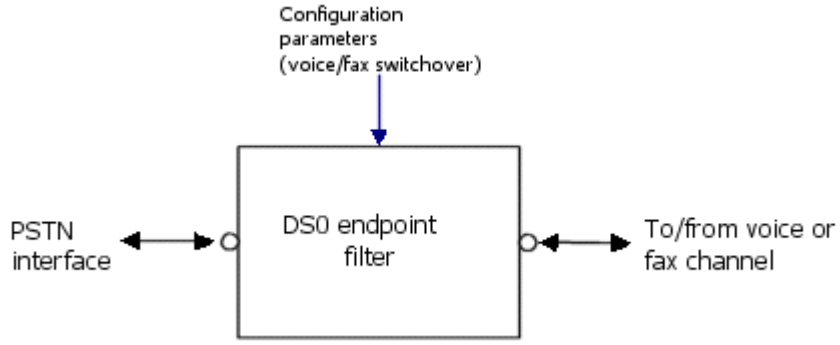
Unsolicited events

None.

Details

Use DS0 endpoint filters to send data to and receive data from PSTN networks. DS0 endpoints are used in both T.38 fax and voice connections.

The following illustration shows a DS0 endpoint filter:



See also

RTP IPv4 full duplex endpoint filter, RTP IPv6 full duplex endpoint filter, T.38 fax relay channel filter, Voice decoder channel filter, Voice encoder channel filter

RTP IPv4 full duplex endpoint filter

Provides an entry and exit point for a duplex voice over IPv4 data stream (data that includes RTP headers within UDP packets) at the data network interface. For information about RTP IPv4 endpoint parameters, refer to RTP IPv4 endpoint address structure and RTP IPv4 endpoint parameter structure.

Filter ID

MSP_ENDPOINT_RTPFDX

Initialization structure

RTPRTCP_ENDPOINT_PARMS

Restrictions

Used in the voice processing connections.

Channel connections

Can connect to full duplex voice channels.

Commands

RTP IPv4 full duplex endpoint filters respond to the following commands:

Command ID	Description
MSP_CMD_RTPFDX_CONFIG	Sends an msp_ENDPOINT_RTPFDX_CONFIG structure that specifies configuration parameters for the endpoint. RTP IPv4 full duplex endpoints must be in a disabled state to receive MSP_CMD_RTPFDX_CONFIG commands.

Command ID	Description
MSP_CMD_RTPFDX_RTCP_EVENTS	Enables or disables sending RTCP reports (disabled by default) to the application through unsolicited events.
MSP_CMD_RTPFDX_LINK_EVENTS	Enables or disables reporting of link availability transitions through unsolicited events.
MSP_CMD_RTPFDX_CHG_QUOTA	Sets the number of frames to be assembled per packet (default is two frames per packet).
MSP_CMD_RTPFDX_MAP	Assigns a payload ID to a vocoder.
MSP_CMD_RTPFDX_DTMF_EVENTS	Specifies how the endpoint responds to RFC 2833 compliant packets. A control value specifies how the endpoint responds to inband DTMF packets. Defined control values include: SEND_NO_EVENTS No MSPEVN_RFC2833_REPORT is sent upon the arrival of an RFC 2833 compliant packet. SEND_FIRST_EVENT MSPEVN_RFC2833_REPORT is sent upon the arrival of the first RFC 2833 compliant packet for a given DTMF digit. SEND_LAST_EVENT MSPEVN_RFC2833_REPORT is sent upon the arrival of the last RFC 2833 compliant packet for a given DTMF digit. SEND_ALL_EVENTS MSPEVN_RFC2833_REPORT is sent upon the arrival of all RFC 2833 compliant packets (limited by the decimation value). A decimation value specifies how many RFC 2833 compliant packets the filter should receive for a particular tone before sending another event. By default SEND_FIRST_EVENT and SEND_LAST_EVENT are enabled.
MSP_CMD_RTPFDX_SEND_2833	Sends an msp_ENDPOINT_RTPFDX_SEND_2833 structure that initiates a sequence of RFC 2833/4733 packets that correspond to a DTMF digit or other tone. Intended for use with Native Play when there is no encoder filter.

RTP IPv4 FDX endpoints can receive commands any time after the endpoint is created, with the exception of the MSP_CMD_RTPFDX_CONFIG command. The RTP FDX endpoint must be disabled to receive MSP_CMD_RTPFDX_CONFIG commands.

When sending MSP_CMD_RTPFDX_CONFIG commands, applications provide a pointer to an msp_ENDPOINT_RTFDX_CONFIG structure. This structure contains all of the parameters used to configure the RTP endpoint when the application created it.

When using the MSP_CMD_RTPFDX_CONFIG command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any endpoint parameters. Otherwise the MSPP service expects a new value for every parameter in the msp_ENDPOINT_RTFDX_CONFIG structure.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Queries

RTP IPv4 full duplex endpoint filters respond to the following query any time after the endpoint is created:

Query ID	Description
MSP_QRY_RTPFDX_STATUS	Returns information about the RTP endpoint filter state in an msp_FILTER_RTPFDX_STATUS structure.

Sending an MSP_QRY_RTPFDX_STATUS query returns the following structure:

```
typedef struct tag_msp_ENDPOINT_RTPFDX_STATUS {

   // RTP Data
   DWORD        FilterId;
   DWORD        ipFilterState;
   IPADDRESS    localIPAddr;
   DWORD        localPort;
   IPADDRESS    remoteIPAddr;
   DWORD        remotePort;
   DWORD        rtcpEvents;
   DWORD        linkEvents;

   // RTP Stats
   DWORD        rxDrop;
   DWORD        rxPkts;
   DWORD        rxBytes;
   DWORD        txDrop;
   DWORD        txFail;
   DWORD        txPkts;
   DWORD        txBytes;

   // RTP Assembler Data
   DWORD        frame_quota;
   DWORD        assemble_dtmfFrames;

   // RTP Disassembler Data
   DWORD        last_pid_rcvd;
   DWORD        mismatches;
   DWORD        last_map_vocoder;
   DWORD        last_map_pid;
   DWORD        disassemble_dtmfFrames;
   DWORD        dtmf_event_control;

} msp_ENDPOINT_RTPFDX_STATUS;
```

The returned query structure provides the following information:

Field	Description
Filterid	Reserved.
ipFilterState	Indicates the current state of the filter and the socket state with the CG board IPv4 stack: Created (0x00) The filter is created. Configured (0x01) The filter is configured with its IP and RTP parameters. Starting (0x02) The filter is enabled and is in the process of opening a socket with the CG board IPv4 stack. Active (0x03) The filter has an open socket and is processing RTP data. Stopping (0x04) The filter is disabled and is in the process of closing its socket with the CG board IPv4 stack. Stop/Destroy (0x05) The filter received a destroy indication and is closing its socket with the CG board IPv4 stack prior to destroying itself. Destroying (0x06) The filter is cleaning up any associated on-board resources before destroying itself.
localIPAddr	Local IPv4 address of the Ethernet interface on which the RTP session runs.
localPort	Local UDP port number associated with the RTP session.
remoteIPAddr	Remote IPv4 address associated with RTP session's destination.
remotePort	Remote UDP port number associated with the RTP session's destination.
rtcpEvents	Indicates whether or not the endpoint is configured to generate RTCP report events.
linkEvents	Indicates whether or not the endpoint is configured to generate route availability events.
rxDrop	Number of RTP/IP packets dropped by the filter when not in an active state.
rxPkts	Number of valid RTP packets received by this filter.

Field	Description
rxBytes	Number of bytes received by this filter. Includes header bytes as well as data bytes.
txDrop	Number of RTP packets that the filter failed to transfer to the data network interface due to internal error conditions.
txFail	Number of socket send errors that occurred.
txPkts	Number of RTP packets delivered to the socket layer of the CG board's IP stack.
frame_quota	Number of media frames assembled in each payload.
assemble_dtmfframes	Number of outbound inband DTMF frames sent.
last_pid_rcvd	The last RTP payload type (identifier) received.
mismatches	The number of received RTP payload types that were not matched to any supported vocoder.
last_map_vocoder	The Dialogic vocoder used in the last map command. Refer to Vocoder filter attributes for a description of the possible values.
last_map_pid	The payload type (identifier) used in the last map command.
disassemble_dtmfFrames	Number of inbound inband DTMF frames received.
dtmf_event_control	Indicates whether or not the endpoint is configured to generate MSPEVN_RFC2833_REPORT when it receives inband DTMF packets.

Unsolicited events

RTP IPv4 full duplex endpoints can return the following unsolicited events:

Event	Description
MSPEVN_2833_SEND_COMPLETE	Indicates that a sequence of packets initiated by the command MSP_CMD_RTPFDX_SEND_2833 is complete.
MSPEVN_RFC2833_REPORT	Indicates that the endpoint received an RFC 2833 compliant packet.
MSPEVN_RTCP_REPORT	Returns session information about an active RTP/RTCP session.

Event	Description
MSPEVN_NOROUTE_AVAILABLE	Indicates that an RTP endpoint lost the ability to route outbound data to its destination.
MSPEVN_ROUTE_AVAILABLE	Indicates that an RTP endpoint formerly incapable of transmitting data is now back in service.

For more information about RTP endpoint unsolicited events or inband DTMF carriage, refer to the *Fusion Developer's Manual*.

Details

RTP payloads can contain more than one frame of encoded media referred to as compound payloads, or more generically, compound packets. For example, the G.723.1 vocoder produces frames that are 30 ms long. An RTP payload, or compound packet, can contain more than one of these frames. If it contained two, there are 60 ms of encoded information (two 30 ms segments packed sequentially).

Receiving data

Packets that the RTP IPv4 endpoint receives from the IPv4 network can contain a variable number of media frames. The endpoint adapts to the type of payload it receives. The frames in one compound packet may contain data from one type of encoder, while frames in the next compound packet contain data from a different type of encoder. RTP IPv4 endpoints separate frames into packets according to the characteristics of the detected encoder and transfers the frames in the same order in which the frames were assembled.

The RTP IPv4 endpoint timestamps output data for all but the first packet from a compound payload. These timestamps are derived from the timestamp in the compound packet's RTP header. The header timestamp is used for the first frame in the input payload.

The RTP header from the IPv4 network specifies the type of encoding in the payload through the payload_id field. The range of values for this field is static and is defined in the audio/video profile for RTP, RFC 1890 (draft 8). Unless otherwise notified, the filter expects these values for the vocoders Dialogic supports. For example, the static payload type value for G723.1 is 4. If the filter receives a value of 4 in the payload_id field, it assumes the payload encoding is G723.1. Applications can reassign payload_ids at any time.

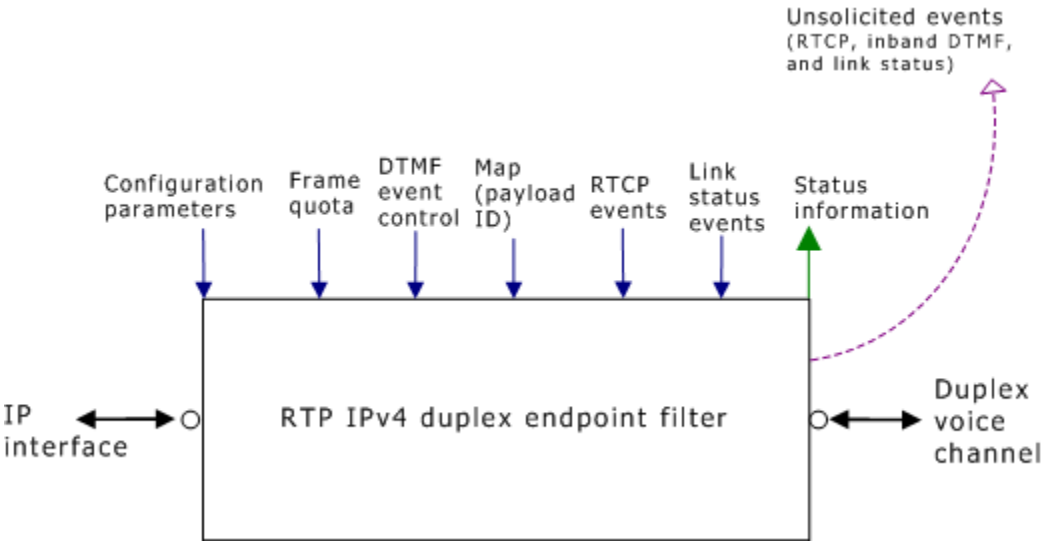
Applications can assign G723.1 a different payload type value from the dynamic range 96 - 127. When this happens, the application must inform the RTP endpoint of the change (using the MSP_CMD_RTPFDX_MAP command) so that the endpoint accepts data associated with a different payload ID. Any vocoder can be assigned a dynamic payload type. If the RTP IPv4 endpoint receives compound packets (that is, packets with multiple frames per packet), the jitter filter depth must be set to a value greater than or equal to the number of frames per packet.

Sending data

The frame_quota parameter specifies the maximum number of media frames the endpoint includes in each payload. This quota should be set to the maximum number of frames that the encoding algorithm allows per payload. Applications can use the MSP_CMD_RTPFDX_CHG_QUOTA filter command to configure the RTP endpoint's frame quota.

The RTP endpoint transfers sequences of packets containing a media header and profile, and a payload containing one or more media frames. The header represents the first frame in the payload.

The following illustration shows an RTP IPv4 full duplex endpoint filter:



See also

DS0 full duplex endpoint filter, Jitter channel filter, RTP switching channel filter, Voice decoder and encoder channel filters

RTP IPv4 simplex receive endpoint filter

Provides an entry point for a simplex voice over IPv4 data stream (data that includes RTP headers within UDP packets) received from the IP interface. For information about RTP IPv4 endpoint parameters, refer to RTP IPv4 endpoint address structure and RTP IPv4 endpoint parameter structure.

Filter ID

MSP_ENDPOINT_RTPIN

Initialization structure

RTPRTCP_ENDPOINT_PARMS

Restrictions

Used in the voice processing connections.

Channel connections

Can connect to simplex voice decode channels.

Commands

RTP IPv4 simplex receive endpoint filters respond to the following commands:

Command ID	Description
MSP_CMD_RTPIN_CONFIG	Sends an msp_ENDPOINT_RTPIN_CONFIG structure that specifies configuration parameters for the endpoint. RTP IPv4 simplex receive endpoints must be in a disabled state to receive MSP_CMD_RTPIN_CONFIG commands.
MSP_CMD_RTPIN_RTCP_EVENTS	Enables or disables sending of RTCP reports to the application through unsolicited events.
MSP_CMD_RTPIN_MAP	Assigns a payload ID to a vocoder.
MSP_CMD_RTPIN_DTMF_EVENTS	Specifies how the endpoint responds to inband DTMF carriage packets. A control value specifies how the endpoint responds to inband DTMF packets. Defined control values include: SEND_NO_EVENTS No MSPEVN_RFC2833_event is sent upon the arrival of an RFC 2833 compliant packet. SEND_FIRST_EVENT MSPEVN_RFC2833_REPORT is sent upon the arrival of the first RFC 2833 compliant packet for a given DTMF digit. SEND_LAST_EVENT MSPEVN_RFC2833_REPORT is sent upon the arrival of the last RFC 2833 compliant packet for a given DTMF digit. SEND_ALL_EVENTS MSPEVN_RFC2833_REPORT event is sent upon the arrival of all RFC 2833 compliant packets (this is limited by the decimation value). A decimation value specifies how many RFC 2833 compliant packets the filter should receive for a particular tone before sending another event. By default SEND_FIRST_EVENT and SEND_LAST_EVENT are enabled.

RTP IPv4 receive endpoints can receive commands any time after the endpoint is created except for the MSP_CMD_RTPIN_CONFIG command. The RTP receive endpoint must be disabled to receive MSP_CMD_RTPIN_CONFIG commands.

When sending MSP_CMD_RTPIN_CONFIG commands, applications provide a pointer to an msp_ENDPOINT_RTPIN_CONFIG structure. This structure contains all of the parameters used to configure the RTP endpoint when the application created it (parameters that do not apply to the specific endpoint type, for example assembler parameters for a simplex receive endpoints, are ignored by the MSPP service).

When using the MSP_CMD_RTPIN_CONFIG command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any endpoint parameters. Otherwise the MSPP service expects a new value for every parameter in the msp_ENDPOINT_RTIN_CONFIG structure.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Queries

RTP IPv4 simplex receive endpoint filters respond to the following query:

Query ID	Description
MSP_QRY_RTPIN_STATUS	Returns information about the RTP endpoint filter state in an msp_FILTER_RTPIN_STATUS structure.

MSP_QRY_RTPIN_STATUS queries return the following structure:

```
typedef struct tag_msp_ENDPOINT_RTPIN_STATUS {

   // RTP Data
   DWORD         FilterId;
   DWORD        ipFilterState;
   IPADDRESS    localIPAddr;
   DWORD        localPort;
   IPADDRESS    remoteIPAddr;
   DWORD        remotePort;
   DWORD        rtcpEvents;
   DWORD        linkEvents;

   // RTP Stats
   DWORD        rxDrop;
   DWORD        rxPkts;
   DWORD        rxBytes;

   // RTP Disassembler Data
   DWORD        last_pid_rcvd;
   DWORD        mismatches;
   DWORD        last_map_vocoder;
   DWORD        last_map_pid;
   DWORD        disassemble_dtmfFrames;
   DWORD        dtmf_event_control;

} msp_ENDPOINT_RTPIN_STATUS;
```

The returned query structure provides the following information:

Field	Description
Filterid	Reserved.

Field	Description
ipFilterState	Indicates the current state of the filter and the socket state with the CG board IPv4 stack: Created (0x00) The filter is created. Configured (0x01) The filter has been configured with its IP and RTP parameters. Starting (0x02) The filter is enabled and is in the process of opening a socket with the CG board IPv4 stack. Active (0x03) The filter has an open socket and is processing RTP data. Stopping (0x04) The filter is disabled and is in the process of closing its socket with the CG board IPv4 stack. Stop/Destroy (0x05) The filter received a destroy indication and is closing its socket with the CG board IPv4 stack prior to destroying itself. Destroying (0x06) The filter is cleaning up any associated on-board resources before destroying itself.
localIPAddr	Local IPv4 address of the Ethernet interface on which the RTP session runs.
localPort	Local UDP port number associated with the RTP session.
remoteIPAddr	Remote IPv4 address associated with the RTP session's destination.
remotePort	Remote UDP port number associated with the RTP session's destination.
rtcpEvents	Indicates whether or not the endpoint is configured to generate RTCP report events.
linkEvents	Indicates whether or not the endpoint is configured to generate route availability events.
rxDrop	Number of RTP/IP packets dropped by the filter when not in an active state.
rxPkts	Number of valid RTP packets received by this filter.
rxBytes	Number of bytes received by this filter. Includes header bytes as well as data bytes.

Field	Description
last_pid_rcvd	The last RTP payload type (identifier) received.
mismatches	The number of received RTP payload types that were not matched to any supported vocoder.
last_map_vocoder	The Dialogic vocoder used in the last map command. Refer to Vocoder filter attributes for a description of the possible values.
last_map_pid	The payload type (identifier) used in the last map command.
disassemble_dtmfFrames	Number of inbound inband DTMF frames received.
dtmf_event_control	Indicates whether or not the endpoint is configured to generate MSPEVN_RTCP_REPORT events when it receives inband DTMF packets.

Unsolicited events

The RTP IPv4 simplex receive endpoints generate the following unsolicited events when the DtmfMode parameter is set:

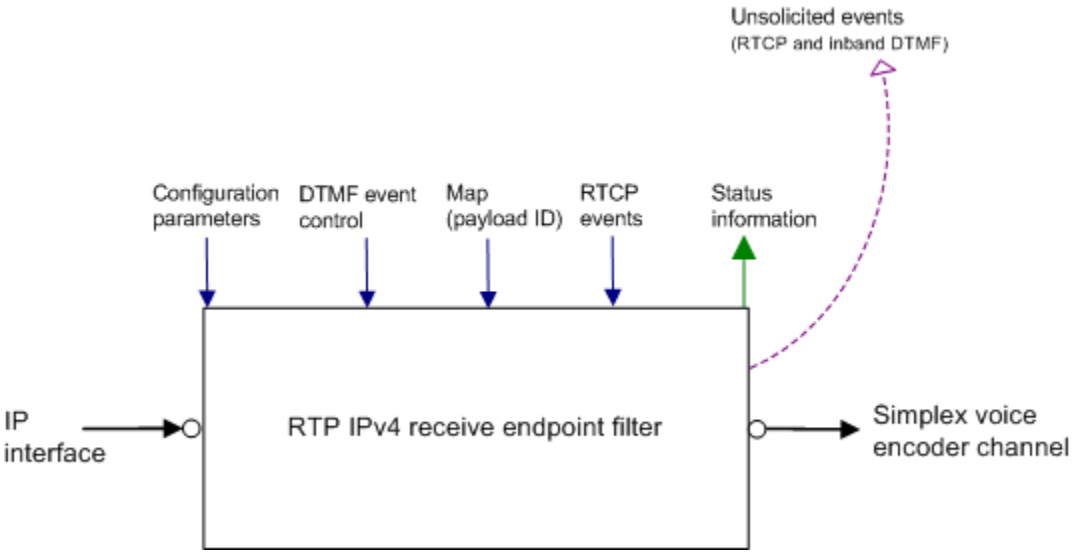
Event	Description
MSPEVN_RTCP_REPORT	When the endpoint is configured to return RTCP information, these events return RFC 1889 compliant RTCP report information.
MSPEVN_RFC2833_REPORT	When the endpoint is configured (during initialization or with the MSP_CMD_RTPIN_DTMF_EVENTS command) to detect inband DTMF packets, these events return information about the tone information received in a DTMF packet.

RTP simplex receive endpoint filters can generate unsolicited MSPEVN_RTCP_REPORT events when they are configured (during creation) to return RTCP information. For more information about these events, refer to MSPP unsolicited events.

Details

For more information about receiving information from a packet network through RTP simplex receive endpoints, refer to the RTP IPv4 full duplex endpoint filter.

The following illustration shows an RTP IPv4 simplex receive endpoint filter:



See Also

DS0 full duplex endpoint filter, Jitter, RTP switching channel filter, Voice decoder channel filter

RTP IPv4 simplex send endpoint filter

Provides an exit point for a simplex voice over IPv4 data stream (data that includes RTP headers within UDP packets) sent to the data network interface. For information about RTP IPv4 endpoint parameters, refer to RTP IPv4 endpoint address structure and RTP IPv4 endpoint parameter structure.

Filter ID

MSP_ENDPOINT_RTPOUT

Initialization structure

RTPRTCP_ENDPOINT_PARMS

Restrictions

Used in the voice processing connections.

Channel connections

Can connect to simplex voice encode channels.

Commands

RTP IPv4 simplex send endpoint filters respond to the following commands:

Command ID	Description
MSP_CMD_RTPOUT_CONFIG	Sends an msp_ENDPOINT_RTPOUT_CONFIG structure that specifies configuration parameters for the endpoint. RTP IPv4 simplex endpoints must be in a disabled state to receive MSP_CMD_RTPOUT_CONFIG commands.
MSP_CMD_RTPOUT_RTCP_EVENTS	Enables or disables sending RTCP reports to the application through unsolicited events.
MSP_CMD_RTPOUT_LINK_EVENTS	Enables or disables reporting of link availability transitions through unsolicited events.
MSP_CMD_RTPOUT_CHG_QUOTA	Sets the number of frames to be assembled per packet (default is two frames per packet).
MSP_CMD_RTPOUT_SEND_2833	Sends an msp_ENDPOINT_RTPOUT_SEND_2833 structure that initiates a sequence of RFC 2833/4733 packets that correspond to a DTMF digit or other tone. Intended for use with Native Play when there is no encoder filter.

RTP IPv4 simplex send endpoints can receive commands any time after the endpoint is created except for the MSP_CMD_RTPOUT_CONFIG command. The RTP send endpoint must be disabled to receive MSP_CMD_RTPOUT_CONFIG commands.

When sending MSP_CMD_RTPOUT_CONFIG commands, applications provide a pointer to an msp_ENDPOINT_RPTOUT_CONFIG structure. This structure contains all of the parameters used to configure the RTP endpoint when the application created it (parameters that do not apply to the specific endpoint type, for example, disassembler parameters for a simplex send endpoints are ignored by the MSPP service).

When using the MSP_CMD_RTPOUT_CONFIG command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any endpoint parameters. Otherwise the MSPP service expects a new value for every parameter in the msp_ENDPOINT_RTPOUT_CONFIG structure.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Queries

RTP IPv4 simplex send endpoint filters respond to the following query:

Query ID	Description
MSP_QRY_RTPOUT_STATUS	Returns information about the RTP endpoint filter state in an msp_FILTER_RTPOUT_STATUS structure.

Sending an MSP_QRY_RTPOUT_STATUS query returns the following structure:

```
typedef struct tag_msp_ENDPOINT_RTPOUT_STATUS {

   // RTP Data
   DWORD        FilterId;
   DWORD        ipFilterState;
   IPADDRESS    localIPAddr;
   DWORD        localPort;
   IPADDRESS    remoteIPAddr;
   DWORD        remotePort;
   DWORD        rtcpEvents;
   DWORD        linkEvents;

   // RTP Stats
   DWORD        txDrop;
   DWORD        txFail;
   DWORD        txPkts;
   DWORD        txBytes;

   // RTP Assembler Data
   DWORD        frame_quota;
   DWORD        assemble_dtmfFrames;

}msp_ENDPOINT_RTPOUT_STATUS;
```

The returned query structure provides the following information:

Field	Description
FilterId	Reserved.
ipFilterState	Indicates the current state of the filter and the socket state with the CG board IPv4 stack: Created (0x00) The filter is created. Configured (0x01) The filter is configured with its IP and RTP parameters. Starting (0x02) The filter is enabled and is in the process of opening a socket with the CG board IPv4 stack. Active (0x03) The filter has an open socket and is processing RTP data. Stopping (0x04) The filter is disabled and is in the process of closing its socket with the CG board IPv4 stack. Stop/Destroy (0x05) The filter received a destroy indication and is closing its socket with the CG board IPv4 stack prior to destroying itself. Destroying (0x06) The filter is cleaning up any associated on-board resources before destroying itself.
localIPAddr	Local IPv4 address of the Ethernet interface on which the RTP session is running.
localPort	Local UDP port number for the RTP session.
remoteIPAddr	Remote IPv4 address associated with the RTP session's destination.
remotePort	Remote UDP port number associated with the RTP session's destination.
rtcpEvents	Indicates whether or not the endpoint is configured to generate RTCP report events.
linkEvents	Indicates whether or not the endpoint is configured to generate route availability events.
txDrop	Number of RTP packets that were dropped due to internal error conditions.
txFail	Number of socket send errors that occurred.

Field	Description
txPkts	Number of RTP packets delivered to the socket layer of the CG board's IP stack.
txBytes	Number of bytes delivered to the socket layer of the CG board's IP stack.
frame_quota	Number of media frames assembled in each payload.
assemble_dtmfFrames	Number of outbound inband DTMF frames sent.

Unsolicited events

RTP IPv4 full simplex send endpoints can return the following unsolicited events:

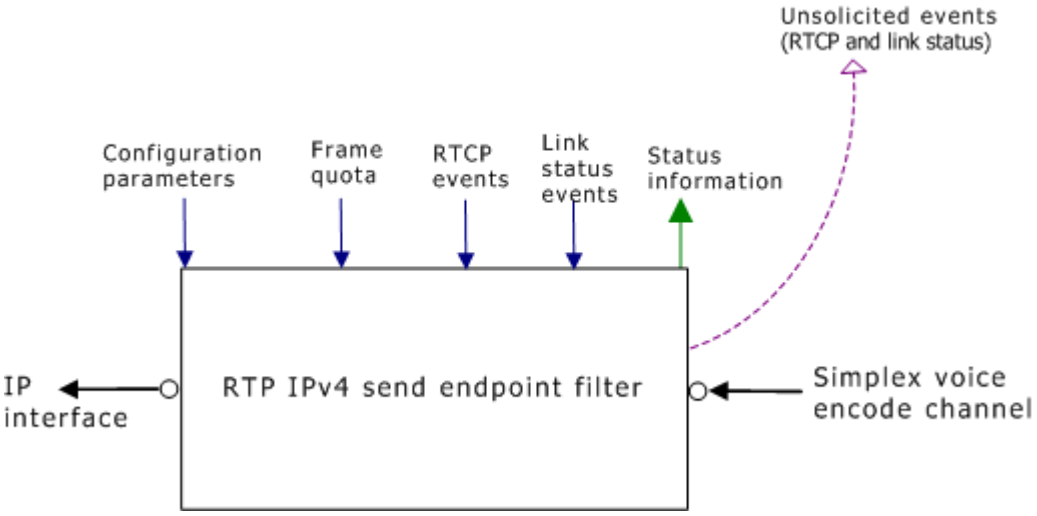
Event	Description
MSPEVN_2833_SEND_COMPLETE	Indicates that a sequence of packets initiated by the command MSP_CMD_RTPOUT_SEND_2833 is complete.
MSPEVN_RTCP_REPORT	Returns session information about an active RTP/RTCP session.
MSPEVN_NOROUTE_AVAILABLE	Indicates that an RTP endpoint lost the ability to route outbound data to its destination.
MSPEVN_ROUTE_AVAILABLE	Indicates that an RTP endpoint formerly incapable of transmitting data is now back in service.

For more information about RTP endpoint unsolicited events or inband DTMF carriage, refer to the *Fusion Developer's Manual*.

Details

For more information about sending information to a packet network through RTP simplex send endpoints, refer to the RTP IPv4 full duplex endpoint filter.

The following illustration shows an RTP IPv4 simplex send endpoint filter:



See also

DS0 full duplex endpoint filter, RTP switching channel filter, Voice encoder channel filter

RTP IPv6 full duplex endpoint filter

Provides entry and exit points for duplex voice over IPv6 data (data that includes RTP headers within UDP packets) at the IP interface. For information about RTP IPv6 endpoint parameters, refer to RTP IPv6 endpoint address structure and RTP IPv6 endpoint parameter structure.

Filter ID

MSP_ENDPOINT_RTPFDX_V6

Initialization structure

RTPRTCP_V6_ENDPOINT_PARMS

Restrictions

Used in the voice processing connections.

Channel connections

Can connect to full duplex voice channels.

Commands

RTP IPv6 full duplex endpoint filters respond to the following commands:

Command ID	Description
MSP_CMD_RTPFDX_V6_CONFIG	Sends an msp_ENDPOINT_RTPFDX_V6_CONFIG structure that specifies configuration parameters for the endpoint. RTP IPv6 full duplex endpoints must be in a disabled state to receive MSP_CMD_RTPFDX_V6_CONFIG commands.

Command ID	Description
MSP_CMD_RTPFDX_RTCP_EVENTS	Enables or disables sending RTCP reports (disabled by default) to the application through unsolicited events.
MSP_CMD_RTPFDX_LINK_EVENTS	Enables or disables reporting of link availability transitions through unsolicited events.
MSP_CMD_RTPFDX_CHG_QUOTA	Sets the number of frames to be assembled per packet (default is two frames per packet).
MSP_CMD_RTPFDX_MAP	Assigns a payload ID to a vocoder.
MSP_CMD_RTPFDX_DTMF_EVENTS	Specifies how the endpoint responds to RFC 2833 compliant packets. A control value specifies how the endpoint responds to inband DTMF packets. Defined control values include: SEND_NO_EVENTS No MSPEVN_RFC2833_REPORT is sent upon the arrival of an RFC 2833 compliant packet. SEND_FIRST_EVENT MSPEVN_RFC2833_REPORT is sent upon the arrival of the first RFC 2833 compliant packet for a given DTMF digit. SEND_LAST_EVENT MSPEVN_RFC2833_REPORT is sent upon the arrival of the last RFC 2833 compliant packet for a given DTMF digit. SEND_ALL_EVENTS MSPEVN_RFC2833_REPORT is sent upon the arrival of all RFC 2833 compliant packets (this is limited by the decimation value). A decimation value specifies how many RFC 2833 compliant packets the filter should receive for a particular tone before sending another event. By default SEND_FIRST_EVENT and SEND_LAST_EVENT are enabled.

RTP IPv6 FDX endpoints can receive commands any time after the endpoint is created with the exception of MSP_CMD_RTPFDX_V6_CONFIG commands. The RTP IPv6 FDX endpoint must be disabled to receive MSP_CMD_RTPFDX_V6_CONFIG commands.

When sending MSP_CMD_RTPFDX_V6_CONFIG commands, applications provide a pointer to an msp_ENDPOINT_RTPFDX_V6_CONFIG structure. This structure contains all of the parameters used to configure the RTP endpoint when the application created it.

When using the MSP_CMD_RTPFDX_V6_CONFIG command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any endpoint parameters. Otherwise the MSPP API expects a new value for every parameter in the msp_ENDPOINT_RTPFDX_V6_CONFIG structure.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Queries

RTP IPv6 full duplex endpoint filters respond to the following query any time after the endpoint is created:

Query ID	Description
MSP_QRY_RTPFDX_V6_STATUS	Returns information about the RTP IPv6 FDX endpoint filter state in an msp_FILTER_RTPFDX_V6_STATUS structure.

Sending an MSP_QRY_RTPFDX_V6_STATUS query returns the following structure:

```
typedef struct tag_msp_ENDPOINT_RTPFDX_V6_STATUS {

   // RTP Data
   DWORD        FilterId;
   DWORD        ipFilterState;
   IPV6ADDR     localIPv6Addr;
   DWORD        localPort;
   IPV6ADDR     remoteIPv6Addr;
   DWORD        remotePort;
   DWORD        rtcpEvents;
   DWORD        linkEvents;
   DWORD        trafficClass;
   DWORD        flowLabel;
   DWORD        linkNum;

   // RTP Stats
   DWORD        rxDrop;
   DWORD        rxPkts;
   DWORD        rxBytes;
   DWORD        txDrop;
   DWORD        txFail;
   DWORD        txPkts;
   DWORD        txBytes;

   // RTP Assembler Data
   DWORD        frame_quota;
   DWORD        assemble_dtmfFrames;

   // RTP Disassembler Data
   DWORD        last_pid_rcvd;
   DWORD        mismatches;
   DWORD        last_map_vocoder;
   DWORD        last_map_pid;
   DWORD        disassemble_dtmfFrames;
   DWORD        dtmf_event_control;

} msp_ENDPOINT_RTPFDX_V6_STATUS;
```

The returned query structure provides the following information:

Field	Description
Filterid	Reserved.
ipFilterState	Indicates the current state of the filter and the socket state with the CG board IPv6 stack: Created (0x00) The filter is created. Configured (0x01) The filter is configured with its IP and RTP parameters. Starting (0x02) The filter is enabled and is in the process of opening a socket with the CG board IPv6 stack. Active (0x03) The filter has an open socket and is processing RTP data. Stopping (0x04) The filter has been disabled and is in the process of closing its socket with the CG board IPv6 stack. Stop/Destroy (0x05) The filter received a destroy indication and is closing its socket with the CG board IPv6 stack prior to destroying itself. Destroying (0x06) The filter is cleaning up any associated on-board resources before destroying itself.
localIPv6Addr	Local IPv6 address (in binary form) of the Ethernet interface on which the RTP session runs.
localPort	Local UDP port number associated with the RTP session.
remoteIPv6Addr	Remote IPv6 address (in binary form) associated with the RTP session's destination.
remotePort	Remote UDP port number associated with the RTP session's destination.
trafficClass	Class or priority of the IPv6 packet.
flowLabel	Specific sequence to which the IPv6 packet belongs.
linkNum	Ethernet interface number (1 or 2).
rtcpEvents	Indicates whether or not the endpoint is configured to generate RTCP report events.

Field	Description
linkEvents	Indicates whether or not the endpoint is configured to generate route availability events.
rxDrop	Number of RTP/IP packets dropped by the filter when not in an active state.
rxPkts	Number of valid RTP packets received by this filter.
rxBytes	Number of bytes received by this filter. Includes header bytes as well as data bytes.
txDrop	Number of RTP packets that were dropped due to internal error conditions.
txFail	Number of socket send errors that occurred.
txPkts	Number of RTP packets delivered to the socket layer of the CG board IP stack.
frame_quota	Number of media frames assembled in each payload.
assemble_dtmfframes	Number of outbound inband DTMF frames sent.
last_pid_rcvd	The last RTP payload type (identifier) received.
mismatches	The number of received RTP payload types that were not matched to any supported vocoder.
last_map_vocoder	The Dialogic vocoder used in the last map command. Refer to Vocoder filter attributes for a description of possible values.
last_map_pid	The payload type (identifier) used in the last map command.
disassemble_dtmfFrames	Number of inbound inband DTMF frames received.
dtmf_event_control	Indicates whether or not the endpoint is configured to generate MSPEVN_RFC2833_REPORT events when it receives inband DTMF packets.

Unsolicited events

RTP IPv6 full duplex endpoints can return the following unsolicited events:

Event	Description
MSPEVN_RFC2833_REPORT	Indicates that the endpoint received an RFC 2833 compliant packet.

Event	Description
MSPEVN_RTCP_REPORT	Returns session information about an active RTP/RTCP session.
MSPEVN_NOROUTE_AVAILABLE	Indicates that an RTP IPv6 endpoint lost the ability to route outbound data to its destination.
MSPEVN_ROUTE_AVAILABLE	Indicates that an RTP IPv6 endpoint formerly incapable of transmitting data is now back in service.

For more information about RTP endpoint unsolicited events or inband DTMF carriage, refer to the *Dialogic® NaturalAccess™ Fusion VoIP API Developer's Manual*.

Details

RTP payloads can contain more than one frame of encoded media referred to as compound payloads, or more generically, compound packets. For example, the G.723.1 vocoder produces frames that are 30 ms long. An RTP payload, or compound packet, can contain more than one of these. If it contained two, there are 60 ms of encoded information (two 30 ms segments packed sequentially).

Receiving data

Packets that the RTP IPv6 endpoint receives from the IP network can contain a variable number of media frames. The endpoint adapts to the type of payload it receives. The frames in one compound packet can contain data from one type of encoder, while frames in the next compound packet contain data from a different type of encoder. RTP IPv6 endpoints separate frames into packets according to the characteristics of the detected encoder and transfers the frames in the same order in which the frames were assembled.

The RTP IPv6 endpoint timestamps transfer data for all but the first packet from a compound payload. These timestamps are derived from the timestamp in the compound packet's RTP header. The header timestamp is used for the first frame in the input payload.

The RTP header from the IP network specifies the type of encoding in the payload through the payload_id field. The range of values for this field is static and is defined in the audio/video profile for RTP, RFC 1890 (draft 8). Unless otherwise notified, the filter expects these values for the vocoders Dialogic supports. For example, the static payload type value for G723.1 is 4. If the filter receives a value of 4 in the payload_id field, it assumes the payload encoding is G723.1. However, applications can reassign payload_ids at any time.

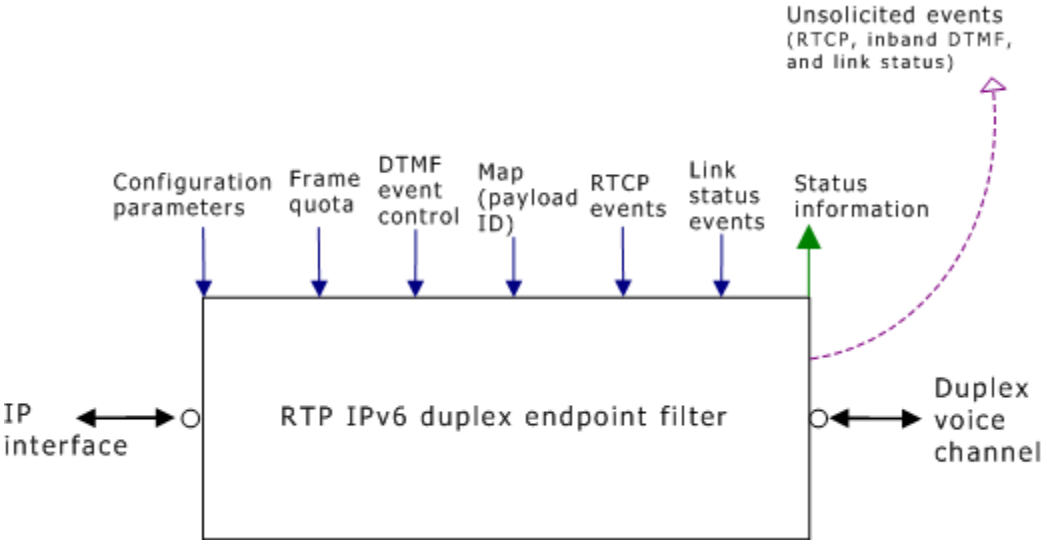
You can assign G723.1 a different payload type value from the dynamic range 96 - 127. In this case, the RTP IPv6 endpoint must be informed of the change (through a MSP_CMD_RTPFDX_MAP command) so that it can accept data with a different payload ID. Any vocoder can be assigned a dynamic payload type. If the RTP IPv6 endpoint receives compound packets (that is, packets containing multiple frames per packet), the jitter filter depth value must be greater than or equal to the number of frames per packet.

Sending data

The frame_quota parameter specifies the maximum number of media frames the endpoint includes in each payload. This quota should be set to the maximum number of frames that the encoding algorithm allows per payload. Applications can use the MSP_CMD_RTPFDX_CHG_QUOTA filter command to configure the RTP endpoint's frame quota.

The RTP endpoint transfers sequences of packets containing a media header and profile, and a payload containing one or more media frames. The header represents the first frame in the payload.

The following illustration shows an RTP IPv6 full duplex endpoint filter:



See also

DS0 full duplex endpoint filter, Jitter channel filter, RTP switching channel filter, Voice decoder and encoder channel filters

RTP IPv6 simplex receive endpoint filter

Provides an entry point for a IPv6 simplex data stream (data that includes RTP headers within UDP packets) received from the IPv6 interface. For more information about RTP IPv6 endpoint parameters, refer to RTP IPv6 endpoint address structure and RTP IPv6 endpoint parameter structure.

Filter ID

MSP_ENDPOINT_RTPIN_V6

Initialization structure

RTPRTCP_V6_ENDPOINT_PARMS

Restrictions

Used in the voice processing connections.

Channel connections

Can connect to simplex voice decode channels.

Commands

RTP IPv6 simplex receive endpoint filters respond to the following commands:

Command ID	Description
MSP_CMD_RTPIN_V6_CONFIG	Sends an msp_ENDPOINT_RTPIN_V6_CONFIG structure that specifies configuration parameters for the endpoint. RTP IPv6 receive endpoints must be in a disabled state to receive MSP_CMD_RTPIN_V6_CONFIG commands.
MSP_CMD_RTPIN_RTCP_EVENTS	Enables or disables sending of RTCP reports to the application through unsolicited events.
MSP_CMD_RTPIN_MAP	Assigns a payload ID to a vocoder.
MSP_CMD_RTPIN_DTMF_EVENTS	Specifies how the endpoint responds to inband DTMF carriage packets. A control value specifies how the endpoint responds to inband DTMF packets. Defined control values include: SEND_NO_EVENTS No MSPEVN_RFC2833_REPORT is sent upon the arrival of an RFC 2833 compliant packet. SEND_FIRST_EVENT MSPEVN_RFC2833_REPORT is sent upon the arrival of the first RFC 2833 compliant packet for a given DTMF digit. SEND_LAST_EVENT MSPEVN_RFC2833_REPORT is sent upon the arrival of the last RFC 2833 compliant packet for a given DTMF digit. SEND_ALL_EVENTS MSPEVN_RFC2833_REPORT is sent upon the arrival of all RFC 2833 compliant packets (this is limited by the decimation value). A decimation value specifies how many RFC 2833 compliant packets the filter should receive for a particular tone before sending another event. By default SEND_FIRST_EVENT and SEND_LAST_EVENT are enabled.

RTP IPv6 receive endpoints can be receive commands any time after the endpoint is created except for the MSP_CMD_RTPIN_V6_CONFIG command. The RTP receive endpoint must be disabled to receive MSP_CMD_RTPIN_V6_CONFIG commands.

When sending MSP_CMD_RTPIN_V6_CONFIG commands, applications provide a pointer to an msp_ENDPOINT_RTFDX_V6_CONFIG structure. This structure contains all of the parameters used to configure the RTP endpoint when the application created it (parameters that do not apply to the specific endpoint type, for example assembler parameters for a simplex receive endpoints, are ignored by the MSPP API).

When using the MSP_CMD_RTPIN_V6_CONFIG command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any endpoint parameters. Otherwise the MSPP API expects a new value for every parameter in the msp_ENDPOINT_RTPFDX_V6_CONFIG structure.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Queries

RTP IPv6 simplex receive endpoint filters respond to the following query:

Query ID	Description
MSP_QRY_RTPIN_V6 STATUS	Returns information about the RTP endpoint filter state in an msp_FILTER_RTPIN_V6_STATUS structure.

MSP_QRY_RTPIN_V6_STATUS queries return the following structure:

```
typedef struct tag_msp_ENDPOINT_RTPIN_V6_STATUS {

   // RTP Data
   DWORD        FilterId;
   DWORD        ipFilterState;
   IPV6ADDR     localIPv6Addr;
   DWORD        localPort;
   IPV6ADDR     remoteIPv6Addr;
   DWORD        remotePort;
   DWORD        rtcpEvents;
DWORD        linkEvents;

   // RTP Stats
   DWORD        rxDrop;
   DWORD        rxPkts;
   DWORD        rxBytes;

   // RTP Disassembler Data
   DWORD        last_pid_rcvd;
   DWORD        mismatches;
   DWORD        last_map_vocoder;
   DWORD        last_map_pid;
   DWORD        disassemble_dtmfFrames;
   DWORD        dtmf_event_control;

} msp_ENDPOINT_RTPIN_V6_STATUS;
```

The returned query structure provides the following information:

Field	Description
Filterid	Reserved.
ipFilterState	Indicates the current state of the filter and the socket state with the CG board IPv6 stack: Created (0x00) The filter is created. Configured (0x01) The filter is configured with its IP and RTP parameters. Starting (0x02) The filter is enabled and is in the process of opening a socket with the CG board IPv6 stack. Active (0x03) The filter has an open socket and is processing RTP data. Stopping (0x04) The filter is disabled, and is in the process of closing its socket with the CG board IPv6 stack. Stop/Destroy (0x05) The filter received a destroy indication, and it is currently in the process of closing its socket with the CG board IPv6 stack prior to destroying itself. Destroying (0x06) The filter is cleaning up all on board resources prior to destroying itself.
localIPv6Addr	IPv6 address (in binary form) of the Ethernet interface on which the RTP session is running.
localPort	Local UDP port number for this RTP session.
remoteIPv6Addr	Remote IPv6 address (in binary form) of the RTP session's destination.
remotePort	Remote UDP port number of the RTP session's destination.
rtcpEvents	Indicates whether or not the endpoint is configured to generate RTCP report events.
linkEvents	Indicates whether or not the endpoint is configured to generate route availability events.
rxDrop	Number of RTP/IP packets dropped by the filter when not in an active state.
rxPkts	Number of valid RTP packets received by this filter.

Field	Description
rxBytes	Number of bytes received by this filter. Includes header bytes as well as data bytes.
last_pid_rcvd	The last RTP payload type (identifier) received.
mismatches	The number of received RTP payload types that were not matched to any supported vocoder.
last_map_vocoder	The Dialogic vocoder used in the last map command. Refer to Vocoder filter attributes for a description of the possible values.
last_map_pid	The payload type (identifier) used in the last map command.
disassemble_dtmfFrames	Number of inbound inband DTMF frames received.
dtmf_event_control	Indicates whether or not the endpoint is configured to generate MSPEVN_RTCP_REPORT events when it receives inband DTMF packets.

Unsolicited events

The RTP IPv6 simplex receive endpoints generate the following unsolicited events when the DtmfMode parameter is set:

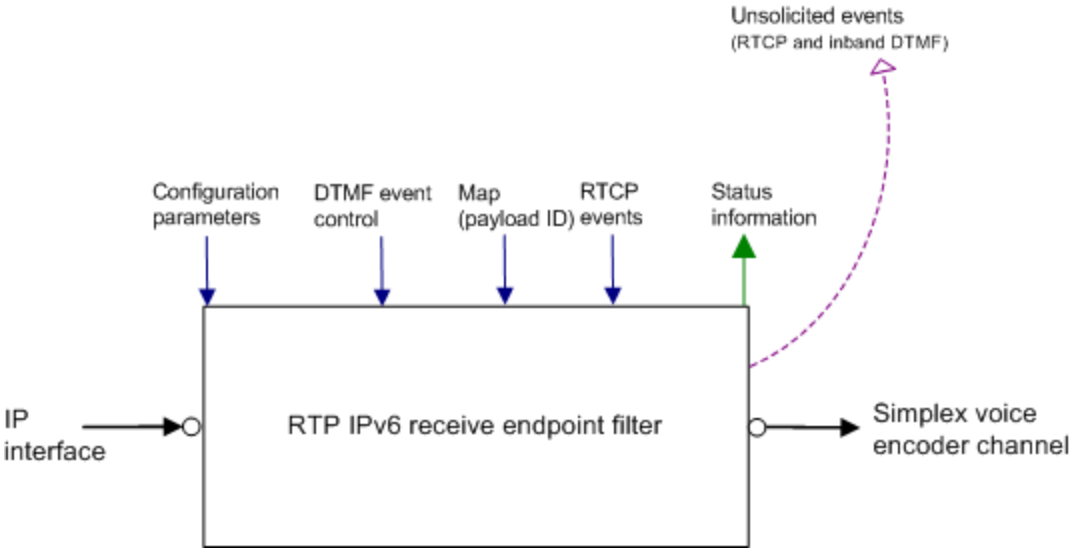
Event	Description
MSPEVN_RTCP_REPORT	When the endpoint is configured to return RTCP information, these events return RFC 1889 compliant RTCP report information.
MSPEVN_RFC2833_REPORT	When the endpoint is configured (during initialization or with the MSP_CMD_RTPIN_DTMF_EVENTS command) to detect inband DTMF packets, these events return information about the tone information received in a DTMF packet.

RTP simplex receive endpoint filters can generate unsolicited MSPEVN_RTCP_REPORT events when they are configured (during creation) to return RTCP information. For more information about these events, refer to MSPP unsolicited events.

Details

For more information about receiving information from a packet network through RTP simplex receive endpoints, refer to the RTP IPv6 full duplex endpoint filter.

The following illustration shows an RTP IPv6 simplex receive endpoint filter:



See Also

DS0 full duplex endpoint filter, Jitter channel filter, RTP switching channel filter, Voice decoder channel filter

RTP IPv6 simplex send endpoint filter

Provide an exit point for simplex voice over IPv6 data stream (data that includes RTP headers within UDP packets) transferred to the IP interface. For information about RTP IPv6 endpoint parameters, refer to RTP IPv6 endpoint address structure and RTP IPv6 endpoint parameter structure.

Filter ID

MSP_ENDPOINT_RTPOUT_V6

Initialization structure

RTPRTCP_V6_ENDPOINT_PARMS

Restrictions

Used in the voice processing connections.

Channel connections

Can connect to simplex voice encode channels.

Commands

RTP IPv6 simplex send endpoint filters respond to the following commands:

Command ID	Description
MSP_CMD_RTPOUT_V6_CONFIG	Sends an msp_ENDPOINT_RTPOUT_V6_CONFIG structure that specifies configuration parameters for the endpoint. RTP IPv6 send endpoints must be in a disabled state to receive MSP_CMD_RTPOUT_V6_CONFIG commands.
MSP_CMD_RTPOUT_RTCP_EVENTS	Enables or disables sending RTCP reports to the application through unsolicited events.
MSP_CMD_RTPOUT_LINK_EVENTS	Enables or disables reporting of link availability transitions through unsolicited events.
MSP_CMD_RTPOUT_CHG_QUOTA	Sets the number of frames to be assembled per packet (default is two frames per packet).

RTP IPv6 send endpoints can receive commands any time after the endpoint is created except for the MSP_CMD_RTPOUT_V6_CONFIG command. The RTP send endpoint must be disabled to receive MSP_CMD_RTPOUT_V6_CONFIG commands.

When sending MSP_CMD_RTPOUT_V6_CONFIG commands, applications provide a pointer to an msp_ENDPOINT_RTOUT_V6_CONFIG structure. This structure contains all of the parameters used to configure the RTP endpoint when the application created it parameters that do not apply to the specific endpoint type, for example, disassembler parameters for a simplex send endpoints are ignored by the MSPP API.

When using the MSP_CMD_RTPOUT_V6_CONFIG command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any endpoint parameters. Otherwise the MSPP API expects a new value for every parameter in the msp_ENDPOINT_RTOUT_V6_CONFIG structure.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* or *mspinit.h* header files.

Queries

RTP simplex send endpoint filters respond to the following query:

Query ID	Description
MSP_QRY_RTPOUT_V6_STATUS	Returns information about the RTP endpoint filter state in an msp_FILTER_RTPOUT_V6_STATUS structure.

Sending an MSP_QRY_RTPOUT_V6_STATUS query returns the following structure:

```
typedef struct tag_msp_ENDPOINT_RTPOUT_V6_STATUS {

   // RTP Data
   DWORD        FilterId;
   DWORD        ipFilterState;
   IPV6ADDR     localIPv6Addr;
   DWORD        localPort;
   IPV6ADDR     remoteIPv6Addr;
   DWORD        remotePort;
   DWORD        rtcpEvents;
   DWORD        linkEvents;
   DWORD        trafficClass;
   DWORD        flowLabel;
   DWORD        linkNum;

   // RTP Stats
   DWORD        txDrop;
   DWORD        txFail;
   DWORD        txPkts;
   DWORD        txBytes;

   // RTP Assembler Data
   DWORD        frame_quota;
   DWORD        assemble_dtmfFrames;

}msp_ENDPOINT_RTPOUT_V6_STATUS;
```

The returned query structure provides the following information:

Field	Description
FilterId	Reserved.
ipFilterState	Indicates the current state of the filter and the socket state with the CG board IP stack: Created (0x00) The filter is created. Configured (0x01) The filter is configured with its IP and RTP parameters. Starting (0x02) The filter is enabled and is in the process of opening a socket with the CG board IPv6 stack. Active (0x03) The filter has an open socket and is processing RTP data. Stopping (0x04) The filter is disabled and is in the process of closing its socket with the CG board IPv6 stack. Stop/Destroy (0x05) The filter received a destroy indication, and it is in the process of closing its socket with the CG board IPv6 stack prior to destroying itself. Destroying (0x06) The filter is cleaning up all on-board resources prior to destroying itself.

Field	Description
localIPv6Addr	IPv6 address of the Ethernet interface this RTP session is running on.
localPort	Local UDP port number for this RTP session.
remoteIPv6Addr	IPv6 address of the destination for this RTP session.
remotePort	UDP port number used at the destination RTP session.
rtcpEvents	Indicates whether or not the endpoint is configured to generate RTCP report events.
linkEvents	Indicates whether or not the endpoint is configured to generate route availability events.
trafficClass	Class or priority of the IPv6 packet.
flowLabel	Specific sequence to which the IPv6 packet belongs.
linkNum	Ethernet interface number (1 or 2).
txDrop	Number of RTP packets that the filter is not able to send out through the IP interface due to internal error conditions.
txFail	Number of socket send errors that occurred.
txPkts	Number of RTP packets delivered to the socket layer of the CG board's IP stack.
txBytes	Number of bytes delivered to the socket layer of the CG board's IP stack.
frame_quota	Number of media frames assembled in each payload.
assemble_dtmfFrames	Number of outbound inband DTMF frames sent.

Unsolicited events

RTP IPv6 full simplex send endpoints can return the following unsolicited events:

Event	Description
MSPEVN_RTCP_REPORT	Returns session information about an active RTP/RTCP session.
MSPEVN_NOROUTE_AVAILABLE	Indicates that an RTP endpoint lost the ability to route outbound data to its destination.

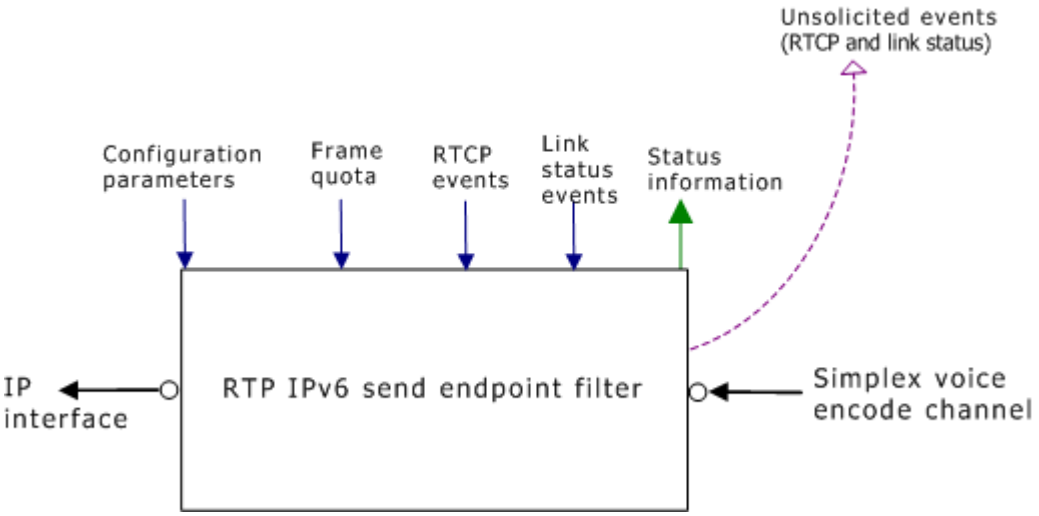
Event	Description
MSPEVN_ROUTE_AVAILABLE	Indicates that an RTP endpoint formerly incapable of transmitting data is now back in service.

For more information about RTP endpoint unsolicited events or inband DTMF carriage, refer to the *Dialogic® NaturalAccess™ Fusion VoIP API Developer's Manual*.

Details

For more information about sending information to a packet network through RTP simplex send endpoints, refer to the RTP IPv6 full duplex endpoint filter.

The following illustration shows an RTP IPv6 simplex send endpoint filter:



See also

DS0 full duplex endpoint filter, RTP switching channel filter, Voice encoder channel filter

T38UDP endpoint filter

Provide entry and exit points for duplex T.38 fax data carried in UDP packets. For more information about T38UDP endpoint filter endpoint filter parameters, refer to T38UDP endpoint filter endpoint address structure and T38UDP endpoint filter endpoint parameter structure.

Note: T38UDP endpoints do not currently support IPv6.

Filter ID

MSP_ENDPOINT_T38FDX

Initialization structure

T38UDP_ENDPOINT_PARMS

Restrictions

Used in T.38 fax full duplex connections.

Channel connections

Connect to T.38 fax full duplex channels.

Commands

T38UDP endpoints respond to the following commands:

Command ID	Description
MSP_CMD_T38UDPTLENC_CONFIG	Configures parameters associated with the endpoint's UDPTL encoder functions.
MSP_CMD_T38UDPTLDEC_CONFIG	Configures parameters associated with the endpoint's UDPTL decoder functions.
MSP_CMD_T38IFPENC_CONFIG	Configures parameters associated with the endpoint's IFP encoder functions.
MSP_CMD_T38IFPDEC_CONFIG	Configures parameters associated with the endpoint's IFP decoder functions.
MSP_CMD_T38UDP_CONFIG	Sets IP parameters in addition to all of the above parameters for the T38UDP endpoint. **Note:** T38UDP endpoints must be in a disabled state when receiving MSP_CMD_T38UDP_CONFIG commands.

With the exception of the MSP_CMD_T38UDP_CONFIG command (see above), T38UDP endpoints can receive filter commands any time after they are created.

When sending MSP_CMD_T38UDP_CONFIG commands, applications provide a pointer to an msp_ENDPOINT_T38UDP_CONFIG structure. This structure contains all the parameters used to configure the T38UDP endpoint when the application created it.

When using the MSP_CMD_T38UDP_CONFIG command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any endpoint parameters. Otherwise, the endpoint expects a new value for every parameter in the structure. For more information about T38UDP endpoint parameters, refer to the Overview of endpoint structures or to the *mspinit.h* header file.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* header file.

Queries

T38UDP full duplex endpoint filters respond to the following query any time after the endpoint is created:

Query ID	Description
MSP_QRY_T38UDP_STATUS	Returns information about the UDP endpoint filter state in an msp_ENDPOINT_T38UDP_STATUS structure.

Sending an MSP_QRY_T38UDP_STATUS query returns the following structure:

```
typedef struct tag_msp_ENDPOINT_T38UDP_STATUS {
//T38 UDP Data
     DWORD      FilterId;
     DWORD      ipFilterState;
     IPADDRESS  localIPAddr;
     DWORD      localPort;
     IPADDRESS  remoteIPAddr;
     DWORD      remotePort;
//T38 UDP Stats
     DWORD      rxDrop;
     DWORD      rxPkts;
     DWORD      rxBytes;
     DWORD      txDrop;
     DWORD      txFail;
     DWORD      txPkts;
     DWORD      txBytes;
//T38 IFP Encoder Data
     DWORD      v21_bytes_per_ifp_packet;
     DWORD      page_modem_ifp_duration;
     DWORD      t30_indicator_send_count;
     DWORD      t30_indicator_send_period;
//T38 UDPTL Encoder Data
     DWORD      error_protection_scheme;
     DWORD      redundant_ifp_packets;
     DWORD      ifp_pkts_per_fec_message;
     DWORD      fec_messages_per_udptl_pkt;
     DWORD      num_duplicate_indicator_udptl_packets;
     DWORD      num_duplicate_v21_udptl_packets;
     DWORD      num_duplicate_page_modem_udptl_packets;
//T38 IFP Decoder Data
     DWORD      queue_size;
     DWORD      buffering_time;
     DWORD      page_data_keep_alive_time;
//T38 UDPTL Decoder Data
     DWORD      enable_error_recovery;
} msp_ENDPOINT_T38UDP_STATUS;
```

Note: FEC is not supported.

The msp_ENDPOINT_T38UDP_STATUS query structure provides the following information:

Field	Description
FilterId	Reserved.

Field	Description
ipFilterState	Indicates the current state of the filter and the socket state with the CG board IP stack: Created (0x00) The filter has been created. Configured (0x01) The filter has been configured with its IP parameters. Starting (0x02) The filter has been enabled and is in the process of opening a socket with the CG board IP stack. Active (0x03) The filter has an open socket and is processing data. Stopping (0x04) The filter has been disabled and is in the process of closing its socket with the CG board IP stack. Stop/Destroy (0x05) The filter has received a destroy indication, and it is currently in the process of closing its socket with the CG board IP stack prior to destroying itself. Destroying (0x06) The filter is cleaning up all on board resources prior to destroying itself.
localIPAddr	Address of the Ethernet interface where this transmission is running.
localPort	Local UDP port number for this transmission.
remoteIPAddr	IP address of the destination for this transmission.
remotePort	UDP port number used at the destination RTP session.
rxDrop	Number of packets dropped by the filter when not in an active state.
rxPkts	Number of valid packets received by this filter.

Field	Description
rxBytes	Number of bytes received by this filter. Includes header bytes as well as data bytes.
txDrop	Number of packets that filter is not able to send out through the IP interface due to internal error conditions.
txFail	Number of socket send errors that occurred.
txPkts	Number of packets delivered to the socket layer of the CG board's IP stack.
txBytes	Number of bytes delivered to the socket layer of the CG board's IP stack.
v21_bytes_per_ifp_packet	Number of V.21 bytes per T.38 IFP packet (default is 1 byte).
page_modem_ifp_duration	Page data time-content in an IFP packet. Time is specified in milliseconds, and is in multiples of 10 (for example, 10, 30 or 60 ms).
t30_indicator_send_count	Maximum number of T.38 indicator packets to transmit to the network.
t30_indicator_send_period	Determines intervals between T.38 indicators transmission to the network. Time is specified in milliseconds, and quantified in multiples of 10 (for example, 10, 90, or 210 ms.).
error_protection_scheme	Type of error protection used by the T.38 fax UDPTL encoder filter. This may be T38UDPTL_EPS_REDUNDANCY.
num_duplicate_v21_udptl_packets	Number of duplicate V.21 UDPTL packets to transmit.
num_duplicate_indicator_udptl_packets	Number of duplicate indicator UDPTL packets to transmit.
num_duplicate_page_modem_udptl_packets	Number of duplicate page modem UDPTL packets to transmit.
redundant_ifp_packets	Number of redundant IFP packets.

Field	Description
queue_size	Size (in number of IFP packets) of the reorder queue used for incoming packet storing, reordering, and error recovering.
buffering_time	Buffering time elapsing before sending data to a T.38 fax modulator/demodulator filter, in order to pre-fill the reordering queue. Time is specified in milliseconds and quantified in multiples of 10 (for example, 30, 90 or 210).
page_data_keep_alive_time	Sets the amount of time to keep a fax session, in the page transmission phase of a fax call, alive in the absence of data from the packet network (that is, absence of data in the IFP reorder queue). The T.4 specification allows the sending of null lines, which alerts the remote fax machine to pause printing until further notice. **Note:** If this parameter is set to a low value or to zero, packet loss in the network may cause the fax session to terminate since the remodulator will drop the carrier when it has no more data to send to the remote fax machine.
enable_error_recovery	Indicates whether the endpoint UDPTL decoder is recovering missing IFPs. • 0x0001= enabled • 0x0000 = disabled

Unsolicited event

When enabled, T38UDP endpoints return the following unsolicited event:

Event	Description
MSPEVN_T38_PACKET_DETECTED	Indicates that the endpoint has received a valid T.38 fax packet.

For information about using T38UPD endpoint unsolicited events to implement voice/T.38 fax switchover, refer to the *Dialogic® NaturalAccess™Fusion VoIP API Developer's Manual*.

Receiving T38 packets from the IP network

When the T38UDP endpoint receives T.38 packets from the packet network interface, it decodes them into primary IFP packets and, if applicable, error recovery information. This process is called T.38 UDPTL decoding. When enabled, T38UDP endpoints generate a MSPEVN_T38_PACKET_DETECTED event the first time that they receive a valid UDPTL packet. This event signals the application to switch over from voice to fax data transmission. When T.38 fax endpoints are disabled and re-enabled, they once again detect the arrival of the first fax packet and return an MSPEVN_T38_PACKET_DETECTED event.

The T38UDP endpoint's IFP decoder receives asynchronous input (IFP packets) from the endpoints UDPTL decoder. The T38UDP endpoint converts data from an asynchronous flow to a synchronous flow before it transfers the data to a T.38 fax full duplex channel. The endpoint reorders the IFP packets during the queuing process and, depending on the error recovery scheme used and the network conditions, lost IFP packets can be recovered from subsequent UDPTL packets.

To prevent underflows in the IFP decoding queue resulting from network jitter, the MSPP service uses a buffering mechanism (controlled by the buffering_time parameter) to pre-fill the queue prior to sending data to the T.38 full duplex channel. This buffering mechanism is similar to the jitter mechanism used by the MSPP voice channel jitter filter.

During unfavorable network conditions, the T38UDP endpoint's efficiency in recovering and reordering IFP packets increases as the number of packets kept in the IFP decoder reorder queue increases. Applications can increase the number of IFP packets in the queue by using the MSP_CMD_T38IFPDEC_CONFIG filter command to increase the value of the buffering_time parameter.

Note: Fax sessions are relatively immune to the delay introduced by IFP decoder buffering mechanism (unlike voice connections where users notice degraded performance when the delay exceeds 200 ms). However, the timing requirements of the T.30 protocol make it nearly impossible to maintain a fax transmission when the one-way delay (roughly, the sum of network delay + buffering time) exceeds 1500 ms.

The actual size of the IFP decoder reorder queue (specified by the queue_size parameter) must be set in accordance with the buffering time. The longer the buffering time, the larger the number of IFP packets that need to be stored in the reorder queue. The queue size should also be large enough to prevent (or limit) the number of overflows during network bursts.

If the IFP decoder queue runs empty during the course of a page transmission (for example, due to bad network conditions), that application can keep the modem signal alive by setting (either through the endpoint's initialization parameters, or through the MSP_CMD_T38IFPDEC_CONFIG filter command) an appropriate page_data_keep_alive_time parameter value. Setting this parameter to a non-zero value specifies that the T.38 fax IFP decoder filter keeps the modem signal alive for a specified (in ms.) duration. This parameter can be particularly useful for networks that frequently experience congestion. This feature only works for non-ECM fax sessions.

Receiving fax data from MSPP T.38 Full Duplex channels

In T.38 fax sessions, fax data and session progress indicators are conveyed through IFP packets. Gateways indicate when they detect signals such as CED, HDLC preamble flags, and modem modulation training by sending T30 indicators. T.38 fax emitting gateways send T30 indicators to receiving gateways and vice versa.

When a T38UDP endpoint receives demodulated data from a T.38 fax full duplex channel, the endpoint analyzes the data and assembles the appropriate IFP packet components depending on the nature of the signal (whether it contains session progress indicators or fax data).

Note: Since V.21 is a low bit rate modem, Dialogic recommends setting the v21_bytes_per_ifp_packet parameter to a low value. For example, a single IFP packet that contains two V.21 bytes, provides a packet period of approximately 2 x 26.6 = 53.3 ms. with the minimum amount of HDLC bit stuffing (2 x 32 = 64 ms. with the maximum amount of HDLC bit stuffing).

For all other modulations (ECM or non-ECM), the size of the page modem data IFP is determined by the page_modem_frames_per_ifp_packet parameter value.

Any parameters that the application sets for the IFP encoder process (either during endpoint initialization or through the MSP_CMD_T38IFPENC_CONFIG endpoint command) will effect the rate at which the T.38 fax gateway sends IFP packets over the IP network.

Before sending T.38 data to the IP network, the T38UDP endpoint encodes T.38 UDPTL packets by adding a primary IFP packet, and optionally adding any error recovery information.

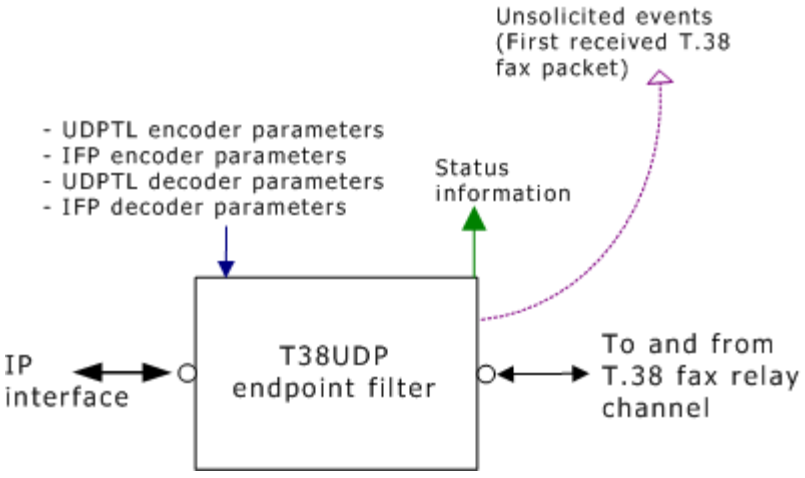
The application specifies the type of T.38 error recovery information the T.38 fax UDPTL encoder filter adds to each T.38 packet by either specifying an appropriate error_protection_scheme parameter the endpoint's initialization parameters, or sending an MSP_CMD_T38UDPTLENC_CONFIG filter command.

To make a T.38 system more tolerant to packet loss, applications can use the UDP packet duplication (this feature is outside the scope of the ITU-T T.38 specification) in conjunction with a T.38 error recovery scheme. However, packet duplication can reduce the throughput of the system by increasing the packet processing load.

UDP packet duplication can be set individually for the following types of primary T.38 IFP packets:

- V21 IFP packets
- Page modem IFP packets
- Indicators

The following illustration shows a T38UDP endpoint:



See also

T.38 fax relay channel filter, Jitter channel filter, Voice decoder channel filter, Voice encoder channel filter

TPKT endpoint filter

Packetizes and multiplexes streams from single channels or multiple channels according to the ThroughPacket algorithm. For more information about TPKT endpoint filter endpoint filter parameters, refer to TPKT endpoint address structure and TPKT endpoint parameter structure.

Note: TPKT endpoints do not currently support IPv6.

Filter ID

MSP_ENDPOINT_TPKT

Initialization structure

TPKT_ENDPOINT_PARMS

Restrictions

Used in full duplex voice connections that use the ThroughPacket multiplexing algorithm.

Filter connections

Can connect to full duplex voice channels.

Commands

TPKT endpoints responds to the following filter command:

Command	Description
MSP_CMD_TPKT_CONFIG	Changes a TPKT endpoint configuration. **Note:** TPKT endpoints must be in a disabled state when receiving MSP_CMD_T38UDP_CONFIG commands.

When applications use the MSP_CMD_TPKT_CONFIG command to send the channel configuration information, they specify information in the following TPKT_ENDPOINT_PARMS structure:

```
typedef struct tag_TPKT_ENDPOINT_PARMS{

   DWORD   localSessionID;
   DWORD   localSessionSeq;
   DWORD   remoteSessionID;
   DWORD   remoteSessionSeq;
   DWORD   deliveryMethod;
   BYTE    remoteGatewayIP[4];
   DWORD   coderType;
   DWORD   frameDuration;
} TPKT_ENDPOINT_PARMS;
```

When sending MSP_CMD_TPKT_CONFIG commands, applications provide a pointer to a TPKT_ENDPOINT_PARMS structure. This structure contains all of the parameters used to configure the ThroughPacket endpoint when the application created it.

When using the MSP_CMD_TPKT_CONFIG command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any endpoint parameters. Otherwise the MSPP service expects a new value for every parameter in the TPKT_ENDPOINT_PARMS structure. For more information about ThroughPacket endpoint parameters, refer to the Overview of endpoint structures or to the *mspinit.h* header file.

For a list of command IDs and their associated structures, refer to the *mspcmd.h* header file.

Queries

TPKT endpoint filters respond to the following filter query any time after the endpoint is created:

Query ID	Description
MSP_QRY_TPKT_CHANNEL	Retrieves information about the TPKT endpoint channel configuration and statistics.

Sending an MSP_QRY_TPKT_CHANNEL query returns the following structure (with substructures):

```
typedef struct tag_msp_TPKT_ENDPOINT_QUERY {

   DWORD FilterId;
   msp_TPKT_CHANNEL_ENTRY
   msp_TPKT_CHANNEL_STATS

} msp_TPKT_ENDPOINT_QUERY;
typedef struct {
   DWORD   rxPkts;
   DWORD   txPkts;
} msp_TPKT_ENDPOINT_STATS;

typedef struct {
   DWORD   channelState;
   DWORD   localSessionID;
   DWORD   remoteSessionID;
   DWORD   localSessionSeq;
   DWORD   remoteSessionSeq;
   DWORD   deliveryMethod;
   BYTE    remoteGatewayIP[4];
   DWORD   coderType;
   DWORD   frameDuration;
   DWORD   secondsActive;
} msp_TPKT_ENDPOINT_ENTRY;
```

The returned msp_TPKT_ENDPOINT_STATS substructure provides the following information:

Field	Description
rxPkts	Number of packets received from the network since the endpoint was enabled.
txPkts	Number of packets sent to the network.

The returned msp_TPKT_ENDPOINT_ENTRY substructure provides the following information:

Field	Description
channelState	State of the session: 0 - created 1 - configured 2 - active
localSessionID	Local session ID.
remoteSessionID	Remote session ID.
localSessionSeq	Sequence number for detecting session overlap: 0|1
remoteSessionSeq	Sequence number for detecting session overlap: 0|1
deliveryMethod	Session delivery method.
remoteGatewayIP	Destination gateway for this session.
coderType	TPKT vocoder type.
frameDuration	Amount of voice data in a packet (for example, 10 ms, 20 ms).
secondsActive	Number of seconds that the TPKT endpoint has been active.

Details

Fusion applications can use MSPP service ThroughPacket (TPKT) endpoints to reduce the total bandwidth needed to transfer multiple sessions of data to a common destination over a packet network. TPKT endpoints group multiple payloads from separate voice channels into larger packets.

To implement TPKT endpoints on a Fusion system, you must:

Step	Action
1	Configure TPKT data transmission parameters in the CG board's keyword file (using the NaturalAccess OAM API configuration parameters) and boot the board. These keywords specify: • TPKT DLM file to load to the CG board. • Data transmission parameters for TPKT endpoints created on the board.
2	Create and configure TPKT endpoints with MSPP service functions.
3	Connect the TPKT endpoints to MSPP channels and enable the channels.

For more information about NaturalAccess OAM API TPKT board keywords, refer to the *Dialogic® NaturalAccess™ Fusion VoIP API Developer's Manual* or the CG board installation manual.

Applications can create MSPP TPKT endpoints in one of two modes:

Protocol	Description
Complex	TPKT transport where payloads from different sessions are combined into larger packets to reduce the number of packets transferred to the network.
Simplex	TPKT transport where media packets for individual sessions are carried in separate packets. This type of transport is similar to RTP and does not significantly effect the bandwidth required by the application.

When the application creates TPKT endpoints, the localSessionID parameter assigns a session identifier to the endpoint. This session ID (an integer in the range of 0 to 4094) remains associated with the endpoint until the application destroys the endpoint. The application uses the remoteSessionID parameter to specify the ID of the session on the remote destination gateway.

Gateways exchange remoteSessionID and remoteSessionSeq information while performing IP call control prior to initiating TPKT sessions.

Applications use the MSPP function **mspSendCommand** (with the MSP_CMD_TPKT_CONFIG command) to change TPKT endpoint configuration parameters, and the **mspSendQuery** function (with the MSP_QRY_TPKT_ENDPOINT query) to retrieve TPKT endpoint configuration parameters. TPKT endpoints can receive commands or queries in any state (that is, enabled or disabled).

localSessionSeq and remoteSessionSeq parameters are flags with a value of 0 or 1. Applications use these parameters to detect session overlap. Session overlap occurs when an application disables an TPKT endpoint and then re-enables the endpoint to send data for a different session. Applications use localSessionSeq and remoteSessionSeq parameters to avoid these overlaps.

For example, when the application creates an active connection using a TPKT endpoint, the application can set the TPKT endpoint's localSessionSeq parameter to 0. If the application disables and then re-enables the TPKT endpoint, it can reset the localSessionSeq parameter to 1 by using **mspSendCommand** (while the endpoint is disabled). The remote TPKT endpoint uses the session sequence flag to discard packets intended for the disabled session. During the transition period when the remote gateway is not yet aware that the session has been disabled, the MSPP service running on the local gateway detects any packets directed to the disabled session and drops them.

TPKT endpoints distinguish between G.723.1 vocoders running at 6.4 kbit/s and G.723.1 vocoders running at 6.4 kbit/s. Standard Fusion G.723.1 vocoders, in accordance with RFC1890, do not distinguish between these vocoder types. When configuring G.723.1 vocoders for 5.33 kbit/s operation, use **mspSendCommand** to change the vocoder payload ID from the default (4) to 127.

For more information about using **mspSendCommand** to change the vocoder payload ID or about using ThroughPacket endpoints refer to the *Fusion Developer's Manual*.

See also

DS0 full duplex endpoint filter, Voice decoder channel filter, Voice encoder channel filter

8. Channel filter reference

Channel filter summary

MSPP channels consist of one or more channel filters, where each filter performs a specific processing task. The MSPP API supports the following channel filters:

Channel filter	Description
Jitter	Performs packet reordering and provides a buffering mechanism for smoothing jitter in packet media streams.
RTP switching	Acts as a pass-through filter to transfer data from one RTP IPv4 or IPv6 endpoint to another.
T.38 fax relay	Carries fax data between DS0 and T38UDP endpoints by performing fax data modulation and demodulation.
Voice decoder	Decodes and decompresses voice data. MSPP voice decoder filters include: • G711 decoder: 64 kbps mu-law or A-law decoder • G723.1 decoder: 6.4/5.3 kbit/s decoder • G726 decoder: 32 kbit/s decoder • G729A decoder: 8.0 kbit/s decoder
Voice encoder	Encodes and compresses voice data. MSPP voice encoder filters include: • G711 encoder: 64 kbit/s mu-law or A-law encoder • G723.1 encoder: 6.4/5.3 kbit/s encoder • G726 encoder: 32 kbit/s encoder • G729A encoder: 8.0 kbit/s encoder

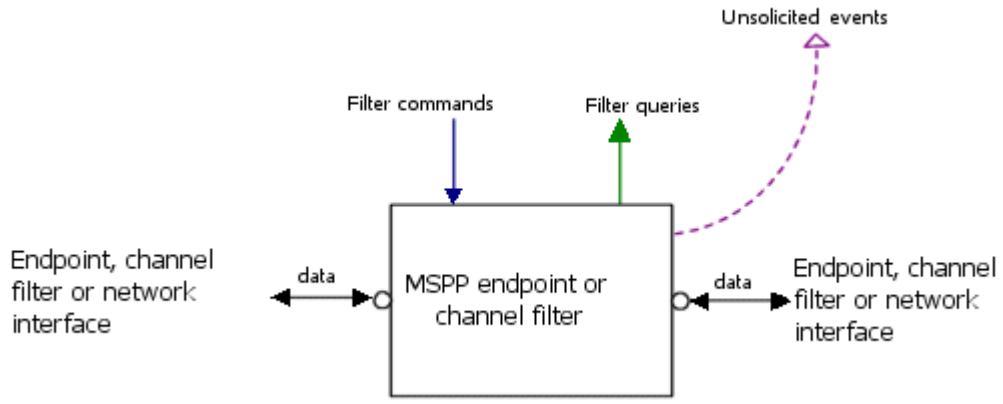
Using the channel filter reference

This section provides a comprehensive, alphabetically-ordered reference to Fusion MSPP filters. A typical filter description includes:

Filter ID	ID for the filter as specified in the *mspcmd.h* file.
Initialization structure	Structure used to specify filter parameters at initialization.
Restrictions	Types of standard MSPP connections that use the filter.
Filter connections	A list of MSPP filters from which the filter can send and receive data.

Filter ID	ID for the filter as specified in the *mspcmd.h* file.
Commands	A list of valid commands that can be sent to the filter with **mspSendCommand**.
Queries	A list of valid queries that can be sent to the filter with **mspSendQuery.**
Unsolicited events	Any filter-specific events returned to the application through the MSPP service.
Details	Detailed information about the MSPP filter and its configuration.
See also	References to related filters.

Each filter description illustrates the filter in the following way:



When applications create MSPP channels with **mspCreateChannel**, the MSPP API automatically creates all of the filters that channel needs to run. The application does not need to individually create and join channel filters.

Jitter channel filter

Receives asynchronous packets and processes them to produce an ordered synchronous output. For information about jitter filter parameters, refer to Voice channel parameter structures.

Filter ID

MSP_FILTER_JITTER

Initialization structure

MSP_VOICE_CHANNEL_PARMS

Restrictions

Used in full duplex voice channels and simplex voice decoder channels.

Filter connections

Valid filter connections include:

Input filters	Output filters
MSP_ENDPOINT_RTPFDX MSP_ENDPOINT_RTPIN MSP_ENDPOINT_RTPFDX_V6 MSP_ENDPOINT_RTPIN_V6	Voice decoder

Commands

Jitter filters respond to the following commands when the associated channel is created:

Command ID	Description	Units	Default
MSP_CMD_JITTER_CHG_DEPTH	Changes jitter depth (default is 2 frames).	Vocoder frames per packet	2
MSP_CMD_JITTER_CHG_ADAPT	Enables or disables the adaptive mode for the jitter filter. Default is disabled.	0 = disable 1 = enable	0 (disabled)
MSP_CMD_JITTER_INIT	Changes any or all jitter filter parameters.	N/A	N/A

When sending MSP_CMD_JITTER_CHG_DEPTH or MSP_CMD_JITTER_CHG_ADAPT jitter filter commands, provide a pointer to the following structure:

```
typedef  struct  tag_msp_FILTER_JITTER_CMD {
              DWORD value;
}  msp_FILTER_JITTER_CMD;
```

The value field contains the new depth or the adaptive mode flag, according to the command. The following constant is defined for jitter commands:

Constant	Description	Value
MSP_JITTER_DEPTH_BOARD_LIMIT	Maximum jitter depth (in vocoder frames) for the jitter depth command. **Note:** MSP_CONST_JITTER_DEPTH_MAX (16) is the maximum recommended value for voice gateways, and is the recommended value for DCE transport.	100

When sending MSP_CMD_JITTER_INIT jitter commands, applications provide a pointer to an msp_FILTER_JITTER_CMD_INIT structure. This structure contains all of the parameters used to configure the jitter filter when the application created the voice channel.

When using the MSP_CMD_JITTER_INIT command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any jitter parameter. Otherwise, the MSPP service expects a new value for every parameter in the msp_FILTER_JITTER_CMD_INIT structure. For more information about jitter filter parameters, refer to the *mspcmd.h* file.

For a list of command IDs and their associated structures, refer to the *mspinit.h* header file.

Jitter filters can receive commands only after the channel has been connected.

Queries

Jitter filters respond to the following query when the associated channel is connected:

Query ID	Description
MSP_QRY_JITTER_GET_STATE	Retrieves information about the jitter filter state. This information is returned in an msp_FILTER_JITTER_STATE structure.

The msp_QRY_JITTER_GET_STATE query structure is defined as follows:

```
typedef struct tag_msp_FILTER_JITTER_STATE {

    DWORD   filter_id;
    DWORD   rx;
    DWORD   rx_accept;
    DWORD   tx;
    DWORD   tx_valid;
    DWORD   requested_depth;
    DWORD   actual_depth;
    DWORD   remote_frame_duration;
    DWORD   underflows;
    DWORD   overflows;
    DWORD   lost_pkts;
    DWORD   duplicate_frames;
    DWORD   reordered_frames;
    DWORD   rejected_frames;
    DWORD   dtmf_frames;
    DWORD   combined_frames;
    DWORD   adapt_enabled;
    DWORD   discarded_frames;
    DWORD   average_depth;

} msp_FILTER_JITTER_STATE;
```

The msp_QRY_JITTER_GET_STATE query structure returns the following information:

Field	Description
filter_id	Reserved.
rx	Number of frames received (before validation).
rx_accept	Number of validated frames received.
tx	Number of frames transmitted (before validation).
tx_valid	Number of validated frames transmitted.

Field	Description
requested_depth	Requested value for the nominal size of the internal queue of this filter. Units are frames. May differ from actual_depth.
actual_depth	Number of frames currently in the jitter filter's internal queue.
remote_frame_duration	Frame size (in milliseconds) of the remote encoder.
underflows	Number of times output occurred when the internal queue was empty.
overflows	Number of packets received that would not fit into the jitter filter's internal queue.
lost_pkts	Number of frames missing at time of output.
duplicate_frames	Number of duplicate frames received.
reordered_frames	Number of frames received out of order.
rejected_frames	Number of frames rejected for any reason.
dtmf_frames	Number of dtmf frames received.
combined_frames	Number of combined frames received.
adapt_enabled	Indicates whether adaptive jitter mode is enabled (1) or disabled (0).
discarded_frames	Number of frames discarded in order to minimize the average jitter depth (adaptive jitter mode only).
average_depth	Average jitter depth value over the previous 5 seconds (adaptive jitter mode only). The displayed value is 100 x the actual value. For example a value of 150 indicates an average jitter depth of 1.5 frames.

Unsolicited events

None.

Details

This filter restores frame ordering and smooths jitter in packet media streams.

Applications use the jitter filter as one part of a simplex or duplex connection (the decode path) from a unique address in an IP network (UDP) to a circuit in a PSTN network. The Jitter accepts frames as its input, which can be out-of-sequence or affected by jitter, and outputs frames in the correct order with reduced jitter. Adjacent frames in the output are spaced by an interval. This interval is determined by the voice decoder in the RTP endpoint, not the jitter filter.

Reordering occurs as the received frames are queued. This queue may underflow (go empty) or overflow because of congestion or bursts from the IP network. Packets that overflow the queue are dropped.

When a voice channel disassembles compound packets (that contain multiple frames per packet), the application must set the jitter depth to greater than or equal to the number of frames per packet. Otherwise, packets are lost and the sound quality may deteriorate.

Jitter depth refers to the number of frames held in the filter queue while the filter performs frame reordering. Applications can send commands to the jitter filter to change its jitter depth. Raising the jitter depth allows the jitter filter to reorder frames more efficiently but also increases the amount of latency introduced into the voice data stream. The application must use a jitter depth that achieves the best balance between efficiency and latency.

When the application enables adaptive mode for the jitter filter, the jitter filter automatically adjusts its jitter depth to the average number of frames required over the previous five seconds. Every five seconds the filter adjusts the jitter depth by raising or lowering the number of frames kept in the filter's internal queue.

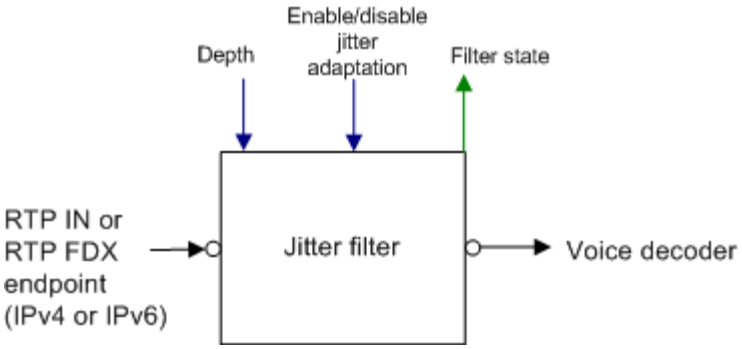
Jitter filter commands control the following characteristics:

Jitter characteristic	Description
Jitter depth	The size (in frames) of the internal queue maintained by a jitter filter. Jitter depth represents a threshold. A jitter filter holds frames in the queue (and does not transmit them) until it accumulates the number of frames specified by the jitter depth. While holding the frames, the jitter filter transfers null frames with empty payloads. Once the number of frames specified by the jitter depth has accumulated, the output function draws from the queue at the rate defined by the vocoder type. In a network with minimal delay or no delay, the number of packets in the jitter queue remains constant. However, if a packet network experiences delays greater than the depth of the jitter queue, the jitter queue may empty. If this occurs, the jitter filter again begins producing null frames while waiting for the specified jitter depth to be reached. Null packets are also produced for missing packets. Null packets are also produced for missing packets.

Jitter characteristic	Description
Adaptive jitter	Automatically and dynamically sets the jitter filter depth according to the frequency with which the jitter filter is receiving frames. For example, if the jitter buffer is receiving an average of six frames per vocoder interval (for example, every 10 ms) over the last five seconds, it adjusts the current size of the jitter queue to 6. When the application enables adaptive jitter (disabled by default), the jitter buffer automatically increases or decreases the number of frames in the jitter depth according to the number of frames received in the previous five seconds.

The requested_depth parameter has a different meaning depending on the adapt_enabled setting. When adaptive jitter is disabled, this parameter indicates the nominal size of the internal queue of the jitter filter. When adaptive jitter is enabled, the requested_depth parameter indicates the current target depth for the jitter queue (rather than the actual number of frames in the queue).

The following illustration shows the position of a jitter filter in a voice channel:



See also

DS0 full duplex, RTP IPv4 full duplex, RTP IPv4 simplex receive, RTP IPv6 full duplex, RTP IPv6 simplex receive, Voice decoder

RTP switching channel filter

Forwards data received from one RTP (IPv4 or IPv6) endpoint to another RTP (IPv4 or IPv6) endpoint.

Filter ID

MSP_FILTER_RTP_SWITCH

Initialization structure

MSP_RTP_SWITCHING_CHANNEL_PARMS

Restrictions

Used in simplex switch connections. Applications can create a duplex voice path by connecting RTP IPv4 and IPv6 endpoints with two simplex switch channels. The endpoints must be enabled before the switching channel is enabled. The switching channel must be disabled before disabling the endpoints.

Filter connections

RTP switch filters can be linked to the following endpoint filters:

Input filters	Output filters
MSP_ENDPOINT_RTPFDX	MSP_ENDPOINT_RTPFDX MSP_ENDPOINT_RTPOUT
MSP_ENDPOINT_RTPIN	MSP_ENDPOINT_RTPFDX MSP_ENDPOINT_RTPOUT
MSP_ENDPOINT_RTPFDX_V6	MSP_ENDPOINT_RTPFDX_V6 MSP_ENDPOINT_RTPOUT_V6
MSP_ENDPOINT_RTPIN_V6	MSP_ENDPOINT_RTPFDX_V6 MSP_ENDPOINT_RTPOUT_V6

Commands

None.

Queries

RTP switching filters respond to the following query when the associated channel is connected:

Query ID	Description
MSP_QRY_RTPSWITCH_GET_STATE	Retrieves information about the switch filter state. This information is returned in an msp_FILTER_RTPSWITCH_STATE structure.

For more information about MSPP service channel initialization and command parameters, refer to the *mspcmd.h* header file.

The msp_FILTER_RTPSWITCH_STATE query structure is defined as follows:

```
typedef struct tag_msp_FILTER_RTPSWITCH_STATE{

    DWORD   FilterId;
    DWORD   ipFilterState;
    DWORD   rxDrop;
    DWORD   rxPkts;
    DWORD   rxBytes;
    DWORD   txDrop;
    DWORD   txFail;
    DWORD   txPkts;
    DWORD   txBytes;

}msp_FILTER_RTPSWITCH_STATE;
```

The msp_FILTER_RTPSWITCH_STATE query structure provides the following information:

Field	Description
filter_id	Reserved.
IpFilterState	State of the filter.
RxDrop	Number of dropped packets.
RxPkts	Number of received packets.
RxBytes	Number of received bytes.
TxDrop	Number of transmit drop packets.
TxFail	Number of failed transmission packets.
TxPkts	Number of transmitted packets.
TxBytes	Number of transmitted bytes.

Unsolicited events

None.

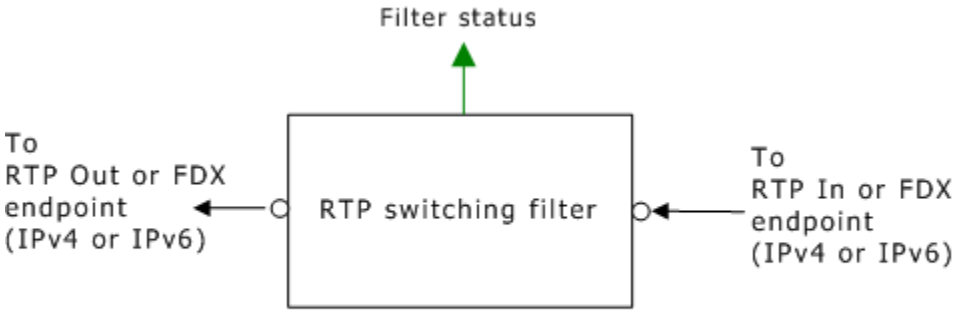
Details

This filter is the only component filter used in a RTP switching channel. RTP switching channels create connections between RTP endpoints so that voice data passes from one endpoint to another in a simplex stream. Applications can create duplex connections by connecting two RTP simplex switch channels to the same RTP endpoint.

When connecting two RTP sessions, consider the following:

- At any given time, there should be only one source of voice for the switch channel. That is, if you switch together two RTP endpoints that are both receiving voice from DS0 endpoints, you must disconnect one of the RTP endpoints from its associated DS0 endpoint. Otherwise, each RTP endpoint simultaneously receives voice data from two network sources.
- All RTP endpoints switched together using RTP switching channels must use the same encoding/decoding algorithm to process voice data. The RTP switching filter only forwards the packets from one RTP endpoint to another - it does not translate the data from one format to another.
- Each RTP endpoint can be connected to a maximum of 32 RTP switching channels.
- When applications connect both an RTP switching channel and a duplex voice channel to the same RTP and DS0 endpoint, they *must* connect the duplex voice channel to the RTP endpoint first. RTP endpoints can support more than one RTP switching connection but can only support one full duplex voice connection (whether the connection consists of a single duplex voice channel or two simplex voice channels).

The following illustration shows an RTP switching filter:



Data in simplex streams must move from RTP In to RTP Out endpoints.

See also

RTP IPv4 full duplex, RTP IPv4 simplex receive, RTP Ipv4 simplex send, RTP IPv6 full duplex, RTP IPv6 simplex receive, RTP IPv6 simplex send

T.38 fax relay channel filter

Demodulates the analog signal coming from the PSTN line and delivers the data to a T38UDP endpoint. This filter also remodulates the appropriate analog signal to be sent to the PSTN based on the information received from a T38UDP endpoint.

Filter ID

MSP_FILTER_FAX_RELAY

Initialization structure

MSP_FAX_CHANNEL_PARMS

Restrictions

Used in T.38 fax full duplex channels.

Filter connections

T.38 fax relay filters connect to DS0 endpoints on the PSTN side of the gateway and T38UDP full duplex endpoints on the IP side of the gateway.

Commands

A T.38 fax relay filter responds to the following command when the associated channel is enabled:

Command ID	Description
MSP_CMD_FAXRELAY_CONFIG	Configures the T.38 fax relay filter.

T.38 fax relay filters must be enabled before receiving commands.

When sending the msp_CMD_FAXRELAY_CONFIG command to a T.38 fax relay filter, include a pointer to the msp_FILTER_FAXRELAY_CONFIG structure. Specify the following parameters for this structure:

Parameter	Description
modemask	Assigns a configuration mask. The following constants are defined for this parameter: • FAXRELAY_MODEMASK_DEFAULT (ECM and NSF enabled) • FAXRELAY_MODEMASK_DISABLE_NSF_FILTERING • FAXRELAY_MODEMASK_DISABLE_ECM • FAXRELAY_MODEMASK_ENABLE_NON_STD_T38_ECM
eventmask	Assigns an event mask that determines the types of unsolicited events the filter returns. The following constants are defined for this parameter (these can be combined using the OR operator together): FAXRELAY_EVENTMASK_DEFAULT All enabled. FAXRELAY_EVENTMASK_REPT_FAX_BEGIN_END Report the beginning and ending of a fax session. FAXRELAY_EVENTMASK_REPT_PASSED_PAGE Report successful passing of a fax page.
maxbitrate	Sets the maximum bit rate of the fax transmission in bits/s: • FAXRELAY_MAXBITRATE_DEFAULT (14400) • FAXRELAY_MAXBITRATE_14400 • FAXRELAY_MAXBITRATE_9600 • FAXRELAY_MAXBITRATE_4800 • FAXRELAY_MAXBITRATE_2400 • FAXRELAY_MAXBITRATE_DISABLE_FILTER
timeout	Sets the session timeout value in ms. The following constants are defined for this parameter: • FAXRELAY_TIMEOUT_DEFAULT (15000 ms) • FAXRELAY_TIMEOUT_MIN (0 ms) • FAXRELAY_TIMEOUT_MAX (65000 ms)

When using the MSP_CMD_FAXRELAY_CONFIG command, applications provide a pointer to an MSP_FAX_CHANNEL_PARMS structure. This structure contains all of the parameters used to configure the T.38 fax relay filter when the application created the T.38 fax channel.

When using the MSP_CMD_FAXRELAY_CONFIG command, applications can specify a value of #FFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any T.38 parameter. Otherwise the filter expects a new value for every parameter in the MSP_FAX_CHANNEL_PARMS structure. For more information about fax relay parameters, refer to the *mspcmd.h* header file.

For a list of command IDs and their associated structures, refer to the *mspinit.h* header file.

Queries

The T.38 fax relay channel filter responds to the following query when the associated channel is enabled:

Query ID	Description
MSP_QRY_FAXRELAY_SESSION_STATUS	Retrieves information about the T.38 relay filter status. This information is returned in a FAXRELAY_SESSION_STATUS structure.

The FAXRELAY_SESSION_STATUS query structure appears as follows:

```
typedef struct tag_FAXRELAY_SESSION_STATUS{
   DWORD         timestamp;
   WORD          bitrate;
   WORD          ecm;
   WORD          ecm_errors;
   WORD          retrain_count;
   WORD          page_count;
   WORD          poll_ecm_errors;
   WORD          poll_retrain_count;
   WORD          poll_page_count;
   WORD          page_modem_count;
   WORD          tcf_count;
   FAXRELAY_STS_FLAGS flags;
} FAXRELAY_SESSION_STATUS;
```

The FAXRELAY_SESSION_STATUS structure provides the following information:

Field	Description
timestamp	Timestamp in milliseconds (10 ms resolution).
bitrate	Last connect rate (2400 - 14400 bits/s).
ecm	Specifies whether fax session used error correction mode.
ecm_errors	If the ecm parameter is set: the number of PPR messages received.
retrain_count	If the ecm parameter is not set: the number of RTN messages received.
page_count	Number of pages passed (emitted or received).
poll_ecm_errors	If the fax session was polled and in ecm mode, the number of PPR messages received in polling mode.

Field	Description
poll_retrain_count	If the fax session was polled and not in ecm mode, the number of RTN messages received in polling mode.
poll_page_count	If the fax session was polled, the number of pages passed (received or emitted).
page_modem_count	Number of times a high speed modem (page modem) was activated (for modulation or demodulation).
tcf_count	Number of times high speed modem (page modem) training was performed (that is, the number of TCFs).
flags	Indicates if certain T.30 messages and other session indicators were ever detected: The following T.30 indicators are monitored: • DIS (digital identification signal) • DCS (digital command signal) • DTC (digital transmit command) • NSF (non-Standard facilities) • CFR (confirmation received) • MCF (message confirmation) • RTP (retrain positive) • RTN (retrain negative) • CRP (command repeat message) • DCN (disconnect message) Other session indicators include: • FCF ever • V.21 frame start • V.21 frame end • V.21 bytes received • HSM ever

Unsolicited events

T.38 fax relay filters return unsolicited events depending on the flags set with the eventmask parameter. The filter can return the following unsolicited events:

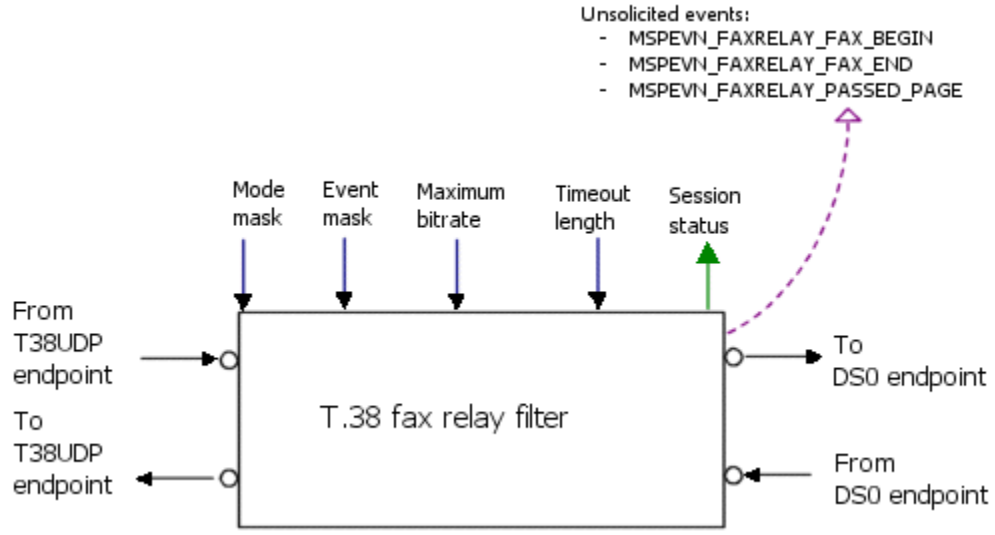
Unsolicited event	Description
MSPEVN_FAXRELAY_FAX_BEGIN	Returned at the beginning of a fax session. This event's data buffer contains a FAXRELAY_FAX_BEGIN structure.
MSPEVN_FAXRELAY_PASSED_PAGE	Returned whenever a page is successfully transferred by the T.38 fax relay channel filter. The event data buffer contains a pointer to a FAXRELAY_PASSED_PAGE structure.
MSPEVN_FAXRELAY_FAX_END	Returned at the end of a fax session. This event's data buffer contains a pointer to the FAXRELAY_SESSION_STATUS structure.

For more information about unsolicited events returned by T.38 fax relay filters, refer to the *mspunsol.h* header file.

Details

The T.38 fax relay filter is the only filter contained within a T.38 fax full duplex channel. A T.38 fax relay filter includes both an encoder (demodulator) and decoder (modulator) component. Only one instance of this filter is used within a full duplex T.38 fax channel to demodulate the analog data received from the DS0 endpoint and to remodulate data received from T38UDP endpoints into analog form.

The following illustration shows a T.38 fax relay filter:



See also

DS0 full duplex, T38UDP

Voice decoder channel filter

Decodes voice data according to a particular algorithm. Parameters within the decoder filter, such as payload ID, IP format and gain, can be altered through filter commands. Decoder filter status information can also be obtained through filter queries. For information about voice encoder filter parameters, refer to Voice channel parameter structures.

Filter ID

The following table lists some of the Fusion decoder filters:

Filter ID	Description	Frame size
MSP_FILTER_G711_DECODER	G.711 64 kbit/s mu-law or A-law decoder	10 ms
MSP_FILTER_G723_DECODER	G.723.1/A 6.4/5.3 kbit/s decoder	30 ms
MSP_FILTER_G726_DECODER	G.726 32 kbit/s decoder	10 ms
MSP_FILTER_G729_DECODER	G.729A/B 8.0 kbit/s decoder	10 ms

Initialization structure

MSP_VOICE_CHANNEL_PARMS

Restrictions

Used in the full duplex voice or simplex voice decoder channels.

Filter connections

Allowed connections for voice decoder filters include:

Input	Output
MSP_FILTER_JITTER	MSP_ENDPOINT_DS0

Commands

Enabled voice decoder filters respond to the following commands when the associated channel is enabled:

Command ID	Description
MSP_CMD_DECODE_MODE	Vocoding mode. Off-line means that the decoder filter plays silence to the DS0 endpoint. 0x0000: off line 0x0001: online Default is online.

Command ID	Description
MSP_CMD_DECODE_DTMF_MODE	Specifies if and how the decoder filter plays DTMF tones when it receives DTMF data (only when inband DTMF carriage is enabled). The application concatenates a control value and a playout value to determine how this works. Defined values for control include: DTMF_DISABLED MSPP channel does not play a DTMF tone when it receives an RFC 2833 compliant inband DTMF carriage packet. The following values can be ORed to specify decoder functionality: • DTMF_INBAND_ENABLED MSPP channel automatically plays a DTMF tone when it detects an RFC 2833 compliant inband DTMF carriage packet. This setting is enabled by default. • DTMF_PLAY_ENABLED MSPP channel plays DTMF tones when the application sends an MSP_CMD_DECODE_PLAY_DTMF_DIGIT command. The playout value is an integer that specifies number of decoder frames for an ongoing tone to continue when no end-of-tone packet is received.
MSP_CMD_DECODE_PLAY_DTMF_DIGIT	Commands the voice decoder filter to play a specified DTMF digit.
MSP_CMD_DECODE_GAIN	Gain value applied to the decoder filter output in S5.10 format. Default is 0x0400 (0 dB). Refer to Calculating vocoder filter gain and loss for a mapping of decibel gain/loss values to linear gain/loss values.
MSP_CMD_DECODE_IP_FORMAT	Specifies the companding algorithm applied to the IP payload (applies to G711 decoder only): 0x0000: mu-law (for T1 systems) 0x0001: A-law (for E1 systems) Default matches system settings.

Command ID	Description
MSP_CMD_DECODE_PAYLOAD_ID	Assigns a payload ID for incoming Fusion media frames. Valid range is 96 - 127. Refer to Vocoder filter attributes for a list of default payload IDs.
MSP_CMD_DECODE_SILENCE_CODE	Specifies the silence pattern sent out by the decoder: 0 – 0xFF (X-law independent) 1 – 0x10 (X-law independent) This command is supported only by the G.711 decoder. All other decoders respond with FUNCTION_NOT_IMPLEMENTED.
MSP_CMD_DECODE_ALL	Resets any or all of the decoder filter parameters.

Note: Voice decoder filters must be enabled when you send them commands.

Caution:	When you change the payload ID of the voice decoder filter, you must also change the payload ID to the same value for the associated RTP endpoint. The voice encoder at the sending gateway must also match this payload ID.

The following table summarizes the commands that available decoder filters support (x = supported):

Decoder	Mode	Gain	IP format	Payload ID	Query	Play DTMF	DTMF mode	Silence code
G.711	x	x	x	x	x	x	x	x
G.723.1	x	x		x	x	x	x	
G.726	x	x	x	x	x	x	x	
G.729A	x	x	x	x	x	x	x	

When sending commands (except for MSP_CMD_DECODE_ALL commands) to decoder filters, include a pointer the following structure:

```
typedef struct tag_msp_FILTER_DECODE_CMD {
    WORD   value;
} msp_FILTER_DECODE_CMD;
```

Use the value field to specify a value for the specified command.

When sending MSP_CMD_DECODE_ALL decoder commands, applications provide a pointer to an msp_FILTER_DECODE_ALL_CMD structure. This structure contains all of the parameters used to configure the decoder filter when the application created the voice channel.

When using the MSP_CMD_DECODE_ALL command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any voice decoder parameter. Otherwise the MSPP service expects a new value for every parameter in the msp_FILTER_DECODE_ALL_CMD structure. For more information about decoder filter parameters, refer to the *mspcmd.h* header file.

For a list of command IDs and their associated structures, refer to the *mspinit.h* header file.

Queries

Voice decoder filters respond to the following queries when the associated channel is enabled:

- **MSP_QRY_SILENCE_CODE**

  Returns the configured silence pattern for the decoder in an msp_FILTER_SILENCE_CODE structure.

  This query is supported only by the G.711 decoder. All other decoders respond with FUNCTION_NOT_IMPLEMENTED.

  Sending an MSP_QRY_SILENCE_CODE query returns the following structure:

```
typedef struct tag_msp_FILTER_SILENCE_CODE {
DWORD  FilterId;
WORD   CommandEcho;
WORD   SilenceCode;
} msp_FILTER_SILENCE_CODE;
```

  The returned query structure provides the following information:

Field	Description
FilterID	Filter ID of the specific decoder filter.
CommandEcho	Return of the query command (0x12).
SilenceCode	Decoder silence pattern (0xFF / 0x10).

- **MSP_QRY_DECODE_STATUS**

  Returns information about the decoder filter state in an msp_FILTER_DECODE_STATE structure. Sending an MSP_QRY_DECODE_STATUS query returns the following structure:

```
typedef struct tag_msp_FILTER_DECODE_STATE{

   DWORD  FilterId;
   WORD   CommandEcho;
   WORD   State;
   WORD   Gain;
   WORD   Rate;
   WORD   PayloadID;
   WORD   IPFormat;
   WORD   DtmfMode;
   WORD   DSPId;
   WORD   ActiveCnt;
   WORD   SIDCnt;
   WORD   NullCnt;
   WORD   DtmfCnt;
   WORD   InvalidPktCnt;
   WORD   DSInQEmptyCnt;
   WORD   DSInQExtraCnt;

} msp_FILTER_DECODE_STATE;
```

The returned query structure provides the following information:

Field	Description
FilterID	Filter ID of the specific decoder filter.
CommandEcho	Return of the query command (0x10).
State	Decoder filter state (online/off line).
Gain	Decoder filter gain.
Rate	Decoder rate.
PayloadId	Expected payload ID of incoming data.
IpFormat	Expected IP format of incoming data (G.711 and G.726 only).
DtmfMode	Whether the voice decoder plays DTMF tones as a result of RFC 2833 compliant packets.
DSPId	DSP number where decoder filter resides.
ActiveCnt	Number of active data packets received.
SIDCnt	Number of SID data packets received.
NullCnt	Number of null data packets received.
DtmfCnt	Number of inband DTMF packets received.
InvalidPktCnt	Number of invalid data packets received.
DSInQEmptyCnt	Number of times the decoder filter received no data.
DSInQExtraCnt	Number of times the decoder filter received more than one data packet.

Unsolicited events

Voice decoder filters generate the following unsolicited event when the DtmfMode parameter is set:

Event	Description
MSPEVN_DPF_DTMF_PLAY_END	When the application commands the decoder to generate a DTMF tone (with the MSP_CMD_DECODE_PLAY_DTMF_DIGIT command), this event signals that the DTMF tone was generated.

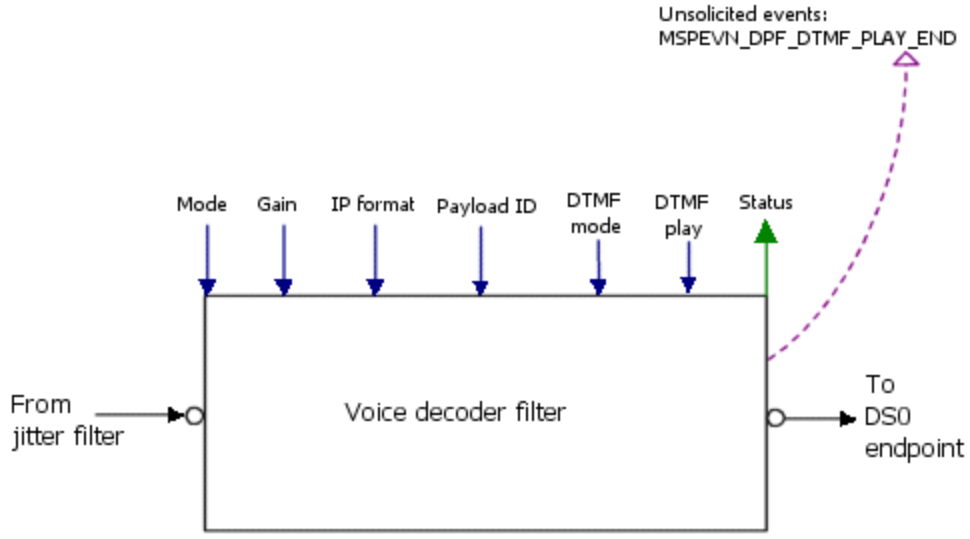
Details

This filter accepts data from a jitter filter, decodes and expands the compressed data, and outputs the resultant PCM signal data to a DS0 endpoint.

Full-duplex voice channels combine voice encoder filters and voice decoder filters to create a two way flow of data. Simplex send channels use voice decoder filters, but without a matching voice encoder filter.

Encoder types and decoder types within a channel do not have to match. Applications can create duplex channels by creating pairs of simplex voice channels that use different vocoding algorithms. For more information about creating this kind of asymmetrical duplex connection, refer to Connecting simplex and duplex channels. Applications can load any mix of encoders and decoders on a CG board (within the constraints of CG board resource management).

The following illustration shows a decoder filter:



See also

DS0 full duplex, Jitter, RTP IPv4 full duplex, RTP IPv4 simplex receive, RTP IPv6 full duplex, RTP IPv6 simplex receive, Voice encoder

Voice encoder channel filter

Encodes voice data according to a particular encoding algorithm. Parameters within the encoder filter, such as payload ID, IP format, volume/gain, and DTMF/CED filter controls can be altered though filter commands. Encoder filter status information can be obtained through a filter query. For information about voice decoder filter parameters, refer to Voice channel parameter structures.

Filter ID

The following table lists some of the Fusion encoder filters:

Filter ID	Description	Frame size
MSP_FILTER_G711_ENCODER	G.711 64 kbit/s mu-law or A-law encoder with or without VAD	10 ms
MSP_FILTER_G723_ENCODER	G.723.1/A 6.4/5.3 kbit/s encoder with or without VAD	30 ms
MSP_FILTER_G726_ENCODER	G.726 32 kbit/s encoder without VAD	10 ms
MSP_FILTER_G729A_ENCODER	G.729A/B 8.0 kbit/s encoder with or without VAD	10 ms

Initialization structure

MSP_VOICE_CHANNEL_PARMS

Restrictions

Used in the full duplex voice or simplex voice encoder channels.

Filter connections

Allowable connections for voice encoder filters include:

Input	Output
MSP_ENDPOINT_DS0 MSP_ENDPOINT_PKTMEDIA MSP_ENDPOINT_MONITOR	MSP_ENDPOINT_RTPFDX MSP_ENDPOINT_RTPOUT MSP_ENDPOINT_RTPFDX_V6 MSP_ENDPOINT_RTPOUT_V6

Commands

Voice encoder filters respond to the following commands when the associated channel is enabled:

Command ID	Description
MSP_CMD_ENCODE_DTMF_MODE	Specifies the mode in which the filter processes DTMF tones. The application concatenates a control value and a payload value to determine how this works. Defined values for control include: DTMF_DISABLED Inband DTMF carriage is not supported for the channel. The following values can be combined using the OR operator to specify encoder functionality: • DTMF_INBAND_ENABLED (default) MSPP channel detects DTMF tones and transfers DTMF information though RFC 2833 compliant packets. • DTMF_EVENTS_ENABLED MSPP channel detects DTMF tones and transfers DTMF information through MSPEVN_DPF_DTMF_REPORT events. • DTMF_SHIFT_ENABLED If set, shifts the timestamp of associated DTMF RTP packets back to better align with the actual start of the DTMF detection. This allows for more synchronized decoder playout of RFC 2833 DTMF digits with respect to the actual DTMF occurrence. • VOICE_ENABLED If DTMF_INBAND_ENABLED is set, the filter sends voice RTP packets as well as RFC 2833 compliant packets for the duration of the DTMF digit. The payload value (range 96-127, default 96) sets the payload ID for inband DTMF packets.
MSP_CMD_ENCODE_DTMFDETECT_PARMS	Modifies the DTMF detection parameters. Requires a pointer to msp_FILTER_ENCODE_DTMFDETECT_PARMS_CMD structure. For more information, refer to DTMF detection parameters.

Command ID	Description
MSP_CMD_ENCODE_PLAY_RFC2833	Plays RFC 2833 RTP packets as specified in the following fields: • Event (WORD) : RFC 2833 event (0 - 255, refer to the RFC 2833 specification for details). • Duration (WORD): Duration of the RFC 2833 event, in milliseconds.
MSP_CMD_ENCODE_MODE	Vocoding mode. Offline means that the encoder produces null media packet frames. Valid entries include: • 0x0000: offline • 0x0001: online Default is online.
MSP_CMD_ENCODE_GAIN	Gain value applied to the encoder filter input in S5.10 format. Default is 0 dB gain (refer to Calculating vocoder filter gain and loss for a table showing gain decibel values and the corresponding linear gain/loss values).
MSP_CMD_ENCODE_VAD	Enables or disables voice activity detection (VAD): • 0x0000: VAD disabled • 0x0001: VAD enabled Default is disabled.
MSP_CMD_ENCODE_SUPPRESS_FILTERS	Enables and/or disables DTMF/CED tone suppression filters. Valid values include: 0x0000: DTMF and CED notch filters disabled 0x0001: DTMF notch enabled, CED notch disabled 0x0002: DTMF notch disabled, CED notch enabled 0x0003: DTMF and CED notch filters enabled Default is 0x0000 (DTMF/CED suppression disabled).

Command ID	Description
MSP_CMD_ENCODE_IP_FORMAT	Specifies the companding algorithm applied to the IP payload (applies to G.711 encoder only): • 0x0000: mu-law (for T1 systems) • 0x0001: A-law (for E1 systems) Default matches system settings.
MSP_CMD_ENCODE_RATE	Vocoding rate. Legal values depend on the encoder. Currently, only G.723.1 supports variable rates. 0 specifies the highest bit rate setting. The bit rate decreases as settings are incremented by one. Therefore for G.723.1, 0 specifies 6.4 kbit/s and 1 specifies 5.3 kbit/s. If a 2 value is defined, it would specify a lower bit rate than 5.3 kbit/s.
MSP_CMD_ENCODE_PAYLOAD_ID	Assigns a payload ID for outgoing media frames. Valid range is 96 - 127. Refer to Vocoder filter attributes for default payload IDs.
MSP_CMD_ENCODE_ALL	Resets any or all of the encoder filter parameters.

Note: Voice encoder filters must be enabled when you send them commands.

Caution:	When you change the payload ID of the voice encoder filter, the receiving gateway must also change the payload ID for the associated voice decoder filter and RTP endpoint.

When sending commands to encoder filters, include a pointer to the following structure:

```
typedef struct tag_msp_FILTER_ENCODE_CMD {
    WORD    value;
} msp_FILTER_ENCODE_CMD;
```

Use the value field to specify a value for the specified command.

When sending MSP_CMD_ENCODE_ALL encoder commands, applications provide a pointer to an msp_FILTER_ENCODE_ALL_CMD structure. This structure contains all the parameters used to configure the encoder filter when the application created the voice channel.

When using the MSP_CMD_ENCODE_ALL command, applications can specify a value of 0xFFFF for 16-bit parameters or 0xFFFFFFFF for 32-bit parameters (or -1 in decimal format) to retain the existing settings for any voice encoder parameter. Otherwise, the MSPP service expects a new value for every parameter in the msp_FILTER_ENCODE_ALL_CMD structure.

The following table shows commands supported by available encoder filter (x = supported):

Encoder	Mode	Gain	VAD	Rate	DTMF/ CED control	Payload ID	IP format	DTMF mode
G.711	x	x	x	x	x	x	x	x
G.723.1	x	x	x	x	x	x		x
G.726	x	x			x	x	x	x
G.729A	x	x	x		x	x		x

For more information about MSPP service channel initialization and command parameters, refer to the *mspcmd.h* header file.

For a list of command IDs and their associated structures, refer to the *mspinit.h* header file.

Queries

Encoder filters respond to the following query when the associated channel is enabled:

Query ID	Description
MSP_QRY_ENCODE_STATUS	Returns information about the voice encoder filter state in an msp_FILTER_ENCODE_STATE structure.

Sending an MSP_QRY_ENCODE_STATUS query returns the following structure:

```
typedef struct tag_msp_FILTER_ENCODE_STATE
{
   DWORD  FilterId;
   WORD   CommandEcho;
   WORD   State;
   WORD   Gain;
   WORD   VadControl;
   WORD   Rate;
   WORD   PayloadID;
   WORD   IPFormat;
   WORD   NotchControl;
   WORD   DtmfMode;
   WORD   DSPId;
   WORD   ActiveCnt;
   WORD   SIDCnt;
   WORD   NullCnt;
   WORD   DtmfCnt;
   WORD   DSOutQFullCnt;
} msp_FILTER_ENCODE_STATE;
```

The returned query structure provides the following information:

Field	Description
Filterid	Filter ID of the encoder filter.
CommandEcho	Returns the filter command to the encoder filter (0x10).
State	Encoder filter state (online/offline).

Field	Description
Gain	Encoder filter gain.
VadControl	VAD control parameter (enabled or disabled).
Rate	Rate of encoded speech (valid only for encoders that support multiple rates). Vocoding rate. Legal values depend on the encoder. Currently, only G.723.1 supports variable rates (where the default is 0 or 6.4 kbit/s). 0 specifies the highest bit rate setting. The bit rate decreases as settings are incremented by one. Therefore for G.723.1, 0 specifies 6.4 kbit/s and 1 specifies 5.3 kbit/s. If a 2 value is defined, it specifies a lower bit rate than 5.3 kbit/s.
PayloadID	Payload ID of outgoing media.
IPFormat	IP format of outgoing G.711 or G.726 media (A-law/mu-law).
NotchControl	DTMF/CED suppression filter parameters.
DtmfMode	Specifies whether or not the encoder is sending DTMF tone information in RFC 2833 compliant packets.
DSPId	DSP number where encode filter resides.
ActiveCnt	Number of active data packets sent.
SIDCnt	Number of SID data packets sent.
NullCnt	Number of null data packets sent.
DtmfCnt	Number of RFC 2833 compliant packets sent.
DSOutQFullCnt	Number of times the encoder filter had no place to output data.

Not all fields are applicable to all encoder filters. Fields that do not apply to a particular filter will return a value that indicates that they do not support it.

Unsolicited events

Voice encoder filters generate the following unsolicited event when the DtmfMode parameter is set:

Event	Description
MSPEVN_DPF_DTMF_REPORT	Returned when the encoder detects a DTMF tone on an incoming data stream.

Details

To create a full-duplex vocoded voice path for a given channel, applications combine voice encoder filters and voice decoder filters within individual channels. The encoder types and the decoder types do not have to match. Applications can also create simplex channels by using decoders without encoders or vice versa.

Voice encoder filters accept PCM signal data and produce a compressed representation of that data. The compressed data is prefixed by a header and sent to a packet engine (RTP endpoint). On CG boards, the packet engine forms an Ethernet/IP/UDP packet from one or more payloads and transmits the fully formed packet to the Ethernet.

The following frequency suppression functions are available for voice encoder filters (these are enabled and disabled through the MSP_CMD_ENCODE_SUPPRESS_FILTERS filter command):

Function	Description
DTMF suppression	Eliminates the high group of DTMF frequencies to help implement DTMF carriage in voice over IP applications.
CED tone suppression	Eliminates the facsimile CED 2100 Hz tone for fax over IP applications. This filter is designed to prevent the 2100 Hz CED tone from reaching the originating end fax machine.

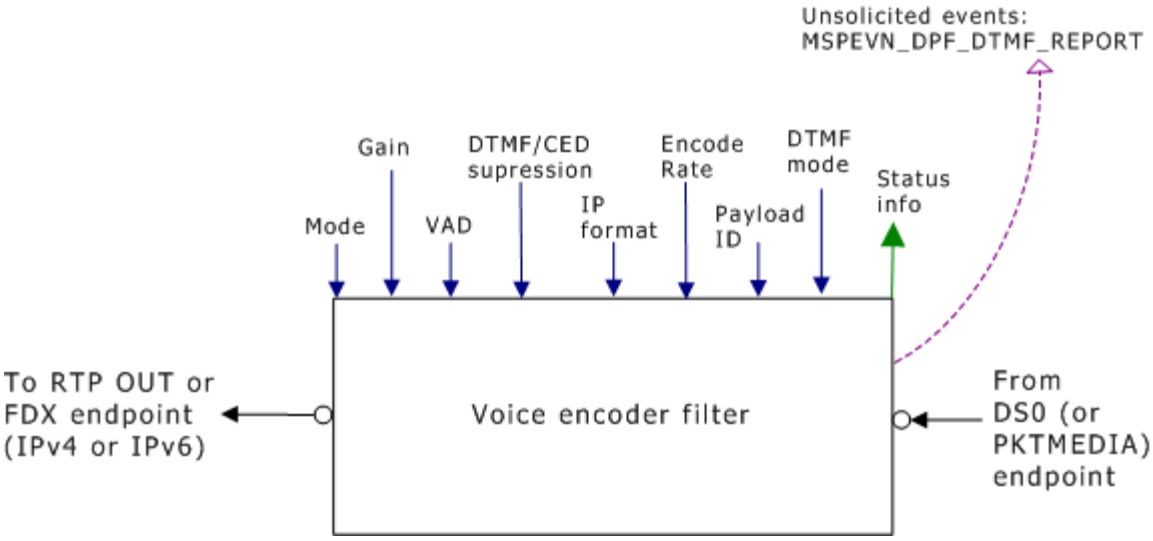
Both tone detection functions are disabled by default, but can be enabled or disabled on the fly. Because the DTMF suppression function suppresses only the high group of DTMF frequencies, callers can still hear the low group of DTMF frequencies on the line.

When using G.723 voice encoder filters, it may be desirable to set the RTP assembler filter's frames-per-packet to 1 to reduce latency. Since G.723 vocoder filters produce frames in 30 ms intervals, setting frames-per-packet to 1 reduces latency by 30 ms.

Caution:	Avoid setting the RTP endpoint frames-per-packet quota to less than 2 when using any encoder filter but G.723.1. Setting the frame quota to less than 2 can exceed the CG board's processing abilities and cause dropped packets. This is because all Fusion encoder filters except G.723.1 produce frames in 10 ms intervals.

Applications can load any mix of encoders and decoders on a CG board within the constraints of CG resource management.

The following illustration shows a voice encoder filter:



DTMF detection parameters

When an encoder is enabled, its dtmf detector parameters are set to default values. The values can be modified by sending an MSP_CMD_ENCODE_DTMFDETECT_PARMS command to the filter while it is enabled. The command must be accompanied by a pointer to the msp_FILTER_ENCODE_DTMFDETECT_PARMS_CMD structure.

All values must be set. Note that there is no capability to query the settings. The initial values that are set when the filter is enabled are shown below.

```
typedef struct tag_msp_FILTER_ENCODE_DTMFDETECT_PARMS_CMD
{
  WORD columnfour;    // 1=detect DTMFs A,B,C,D; 0=don't
  WORD onqualampl;    // min input lev to qual tone
  WORD onthreshold;   // Talkoff versus Cutthrough qualify level
  WORD onqualtime;    // qualify time of DTMF
  WORD offqualampl;   // min input lev of valid DTMF
  WORD offthreshold;  // Hysteresis for talkoff versus cutthrough
  WORD offqualtime;   // disqualify time for tone (ms)
} msp_FILTER_ENCODE_DTMFDETECT_PARMS_CMD;
```

Field values are in proprietary units. The following table describes the conversion from dB or milliseconds, and the useful range of values for each field.

Field	Default	Min	Max	Equivalent	Min	Max	Notes
columnfour	1	0	1				
onqualampl	504	128	16000	-39 dBm0	-51	-9	Linear scale where 0 dBm0 is 45632.
onthreshold	276	173	347	-2.0 dB	-6.0	0	Linear scale where 0 dB is 347.

Field	Default	Min	Max	Equivalent	Min	Max	Notes
onqualtime	28	0	32767	50 ms	22	32789	Time in milliseconds minus 22.
offqualampl	252	128	16000	-45 dBm0	-51	-9	Linear scale where 0 dBm0 is 45632.
offthreshold	200	173	347	-4.8 dB	-6.0	0	Linear scale where 0 dB is 347.
offqualtime	35	0	32767	40 ms	5	32772	Time in milliseconds minus 5.

Use the following formula to convert dB input values to linear:

```
linear= 10**(dB/20) * <0 dB value>
```

For example, using 'C' code:

```
    int dBm0 = -39;
    unsigned onqualampl = pow (10, (double)dBm0/20) * 45632;

    double dB = -2.0
    unsigned onthreshold = pow (10, dB/20) * 347;

    int msec = 50;
    unsigned onqualtime = msec - 22;
```

By comparison, the ADI dtmf detector API (**adiStartDTMF**) performs the above conversions for qualification amplitude and time, but not for the threshold values. You can convert ADI values for offthreshold and onthreshold by dividing by 188.

Example

The default value of ADI.DTMFDETECT.onthreshold is 0xcab0 or 51888. The equivalent encoder value is 51888/188 = 276.

See Also

Jitter, RTP IPv4 full duplex, RTP IPv4 simplex send, RTP IPv6 full duplex, RTP IPv6 simplex send, Voice decoder

9. Errors

Alphabetical error summary

This section provides a summary of MSPP service error codes. The error codes are presented in two tables:

- Alphabetical, by error code name, including a description of the problem and a possible solution
- Numerical, by hexadecimal value, decimal value, and error name

All MSPP service functions return a status code. If the return code is not SUCCESS (0), it is an error code indicating that the function has failed and a reason for the failure.

MSPP service error codes are defined in the include files *mspdef.h* and *ctaerr.h*. The error codes are prefixed with MSPERR_ or CTAERR_.

The following table alphabetically lists MSPP service errors (all errors are 32 bits):

Error name	Hex	Decimal	Description
MSPERR_CANNOT_OPEN_SERVICE_ COMPONENT	0x001B0003	1769475	Cannot open a sub-component of the service.
MSPERR_CHANNEL_NOT_CONNECTED	0x001B0006	1769478	MSPP channel is not connected to any endpoints.
MSPERR_CHANNEL_NOT_DEFINED	0x001B0005	1769477	MSPP channel name is not defined.
MSPERR_DRIVER_COMMAND_FAILED	0x001B0008	1769480	SPI command failed.
MSPERR_ENDPOINT_BUSY	0x001B000C	1769484	Command arrived while servicing previous command.
MSPERR_ENDPOINT_NOT_ALLOCATED	0x001B0007	1769479	MSPP endpoint is defined, but not yet allocated on the board.
MSPERR_FILTER_BUSY	0x001B000D	1769485	Command arrived while servicing previous command.
MSPERR_INTERNAL_HANDLE	0x001B0002	1769474	Invalid MSPP channel or endpoint.
MSPERR_INVALID_CONNECTION	0x001B0001	1769473	Connection cannot be found.

Error name	Hex	Decimal	Description
MSPERR_INVALID_HANDLE	0x001B0004	1769476	Invalid MSPP channel or endpoint.
MSPERR_SENDCOMMAND_FAILED	0x001B000A	1769482	**mspSendCommand** returned an error.
MSPERR_SENDQUERY_FAILED	0x001B000B	1769483	**mspSendQuery** returned an error.
MSPERR_UNKNOWN_FILTER_OBJECT	0x001B0009	1769481	The command specified a filter that is not in the channel.

Numerical error listing

The following table numerically lists MSPP service errors:

Hex	Decimal	Error name
0x001B0001	1769473	MSPERR_INVALID_CONNECTION
0x001B0002	1769474	MSPERR_INTERNAL_HANDLE
0x001B0003	1769475	MSPERR_CANNOT_OPEN_SERVICE_COMPONENT
0x001B0004	1769476	MSPERR_INVALID_HANDLE
0x001B0005	1769477	MSPERR_CHANNEL_NOT_DEFINED
0x001B0006	1769478	MSPERR_CHANNEL_NOT_CONNECTED
0x001B0007	1769479	MSPERR_ENDPOINT_NOT_ALLOCATED
0x001B0008	1769480	MSPERR_DRIVER_COMMAND_FAILED
0x001B0009	1769481	MSPERR_UNKNOWN_FILTER_OBJECT
0x001B000A	1769482	MSPERR_SENDCOMMAND_FAILED
0x001B000B	1769483	MSPERR_SENDQUERY_FAILED
0x001B000C	1769484	MSPERR_ENDPOINT_BUSY
0x001B000D	1769485	MSPERR_FILTER_BUSY

Numerical error summary

The following table numerically lists MSPP API errors:

Hex	Decimal	Error name
0x001B0001	1769473	MSPERR_INVALID_CONNECTION
0x001B0002	1769474	MSPERR_INTERNAL_HANDLE
0x001B0003	1769475	MSPERR_CANNOT_OPEN_SERVICE_COMPONENT
0x001B0004	1769476	MSPERR_INVALID_HANDLE
0x001B0005	1769477	MSPERR_CHANNEL_NOT_DEFINED
0x001B0006	1769478	MSPERR_CHANNEL_NOT_CONNECTED
0x001B0007	1769479	MSPERR_ENDPOINT_NOT_ALLOCATED
0x001B0008	1769480	MSPERR_DRIVER_COMMAND_FAILED
0x001B0009	1769481	MSPERR_UNKNOWN_FILTER_OBJECT
0x001B000A	1769482	MSPERR_SENDCOMMAND_FAILED
0x001B000B	1769483	MSPERR_SENDQUERY_FAILED
0x001B000C	1769484	MSPERR_ENDPOINT_BUSY
0x001B000D	1769485	MSPERR_FILTER_BUSY

10. Events

MSPP events

In response to commands executed by an application, the MSPP API generates events and reason codes that indicate the occurrence (or lack) of certain conditions or state changes. This section summarizes the events and reason codes generated by the MSPP API:

- Alphabetical event summary
- Numerical event summary

A DONE event is a NaturalAccess event informing the application that an asynchronous function has completed processing. DONE event codes are in the form MSPEVN_***function***_DONE where ***function*** is the completed function (CREATE_CHANNEL, DESTROY_CHANNEL). DONE events have no special physical or processing characteristics. They have the same physical structure and are retrieved identically to other events.

Alphabetical event summary

The following table alphabetically lists MSPP API events:

Event name	Description
MSPEVN_CONNECT_DONE	Media channel is connected to endpoints.
MSPEVN_CREATE_CHANNEL_DONE	Media channel is created.
MSPEVN_CREATE_ENDPOINT_DONE	MSPP endpoint is created.
MSPEVN_DESTROY_CHANNEL_DONE	Specified media channel is destroyed.
MSPEVN_DESTROY_ENDPOINT_DONE	Specified endpoint is destroyed.
MSPEVN_DISABLE_CHANNEL_DONE	Data flow through specified channel is halted.
MSPEVN_DISABLE_ENDPOINT_DONE	Data flow between specified endpoints is halted.
MSPEVN_RFC2833_REPORT	Unsolicited event returned when an RTP FDX or RTP receive endpoint receives an inband RFC2833 packet.
MSPEVN_DISCONNECT_DONE	Media channel is disconnected from endpoints.
MSPEVN_DPF_DTMF_PLAY_END	Unsolicited event from voice decoder filter reporting that a DTMF tone was generated.
MSPEVN_DPF_DTMF_REPORT	Unsolicited event from voice encoder reporting that a DTMF tone was detected.
MSPEVN_ENABLE_CHANNEL_DONE	Data flow through specified channel is enabled.

Event name	Description
MSPEVN_ENABLE_ENDPOINT_DONE	Data flow between connected MSPP endpoints is enabled.
MSPEVN_FAXRELAY_FAX_BEGIN	Unsolicited event.
MSPEVN_FAXRELAY_FAX_END	Unsolicited event generated by a T.38 fax relay channel filter to indicate the beginning of a T.38 fax session.
MSPEVN_FAXRELAY_PASSED_PAGE	Unsolicited event generated by a T.38 fax relay channel filter to indicate the beginning of a T.38 fax session.
MSPEVN_ROUTE_AVAILABLE	Unsolicited event indicating that an RTP or UDP session formerly incapable of transmitting data is now back in service.
MSPEVN_NOROUTE_AVAILABLE	Unsolicited event indicating that a RTP/UDP session has lost the ability to route outbound data to its destination.
MSPEVN_QUERY_DONE	The specified MSPP query was performed.
MSPEVN_RTCP_BAD_REPORT	Unsolicited event indicating that an RTP endpoint received a malformed RTCP report.
MSPEVN_RTCP_REPORT	Unsolicited event returning RTCP report information received through an RTP endpoint.
MSPEVN_SENDCOMMAND_DONE	The specified MSPP command was sent.
MSPEVN_T38_PACKET_DETECTED	Unsolicited event returned by a T38 UDP endpoint when it receives its first T.38 fax packet.

Numerical event summary

The following table lists events in numerical order:

Hex	Decimal	Event name
0x001B2001	1777665	MSPEVN_CREATE_ENDPOINT_DONE
0x001B2002	1777666	MSPEVN_DESTROY_ENDPOINT_DONE
0x001B2003	1777667	MSPEVN_ENABLE_ENDPOINT_DONE
0x001B2004	1777668	MSPEVN_DISABLE_ENDPOINT_DONE

Hex	Decimal	Event name
0x001B2005	1777669	MSPEVN_CREATE_CHANNEL_DONE
0x001B2006	1777670	MSPEVN_DESTROY_CHANNEL_DONE
0x001B2007	1777671	MSPEVN_ENABLE_CHANNEL_DONE
0x001B2008	1777672	MSPEVN_DISABLE_CHANNEL_DONE
0x001B2009	1777673	MSPEVN_CONNECT_DONE
0x001B200A	1777674	MSPEVN_DISCONNECT_DONE
0x001b2100	1777920	MSPEVN_SENDCOMMAND_DONE
0x001B2200	1778176	MSPEVN_QUERY_DONE
0x001B2300	1778432	MSPEVN_RTCP_REPORT
0x001B2302	1778434	MSPEVN_T38_PACKET_DETECTED
0x001B2303	1778435	MSPEVN_FAXRELAY_FAX_BEGIN
0x001B2304	1778436	MSPEVN_FAXRELAY_FAX_END
0x001B2305	1778437	MSPEVN_FAXRELAY_PASSED_PAGE
0x001B2306	1778438	MSPEVN_RTCP_BAD_REPORT
0x001B2307	1778439	MSPEVN_NOROUTE_AVAILABLE
0x001B2308	1778440	MSPEVN_ROUTE_AVAILABLE
0x001B2309	1778441	MSPEVN_DPF_DTMF_REPORT
0x001B230A	1778442	MSPEVN_RFC2833_REPORT
0x001B230C	1778444	MSPEVN_DPF_DTMF_PLAY_END

Reason codes

Reason codes are generated when a DONE event is given, but an associated task has failed. Reason codes are in the form MSPRSN_FAILED_TO_***function*** where ***function*** is the failed task.

Alphabetical reason code summary

The following table alphabetically lists MSPP service reason codes and their descriptions:

Reason code	Description
MSPRSN_FAILED_TO_ALLOCATE	The MSPP service is unable to allocate on-board resources.
MSPRSN_FAILED_TO_DEALLOCATE	The MSPP service is unable to de-allocate on-board resources.
MSPRSN_FAILED_TO_INTERCONNECT	The MSPP service is unable to connect the channel to an endpoint.
MSPRSN_FAILED_TO_INTERDISCONNECT	The MSPP service is unable to disconnect the channel from an endpoint.
MSPRSN_FAILED_TO_INTRACONNECT	The MSPP service is unable to connect filters within the channel.
MSPRSN_FAILED_TO_INTRADISCONNECT	The MSPP service is unable to disconnect filters within the channel.
MSPRSN_FAILED_TO_START	The MSPP service is unable to start the on-board resource.
MSPRSN_FAILED_TO_STOP	The MSPP service is unable to stop the on-board resource.

Numerical reason code summary

The following table lists MSPP reason codes in numerical order:

Hex	Decimal	Event name
0x001B1001	1773569	MSPRSN_FAILED_TO_ALLOCATE
0x001B1002	1773570	MSPRSN_FAILED_TO_DEALLOCATE
0x001B1003	1773571	MSPRSN_FAILED_TO_START
0x001B1004	1773572	MSPRSN_FAILED_TO_STOP
0x001B1005	1773573	MSPRSN_FAILED_TO_INTRACONNECT
0x001B1006	1773574	MSPRSN_FAILED_TO_INTRADISCONNECT
0x001B1007	1773575	MSPRSN_FAILED_TO_INTERCONNECT
0x001B1008	1773576	MSPRSN_FAILED_TO_INTERDISCONNECT

11. Channel parameters

Overview of channel structures

When creating MSPP channels with **mspCreateChannel**, you must specify channel configuration information in two structures:

- MSPP channel address structure
- MSPP channel parameters structure

Channel address structure

When applications create MSPP channels with **mspCreateChannel**, they specify the following parameters in the MSP_CHANNEL_ADDR structure:

```
typedef struct tag_MSP_CHANNEL_ADDR
{
    DWORD            size;
    DWORD            nBoard;
    MSP_CHANNEL_TYPE channelType;
    DWORD            FilterAttribs;
} MSP_CHANNEL_ADDR;
```

MSP_CHANNEL_ADDR

Dependent function: mspCreateChannel

Field	Type	Default	Units	Description
size	DWORD	N/A	N/A	Size of the address structure.
nBoard	DWORD	N/A	N/A	Channel location (board number).

Field	Type	Default	Units	Description
channelType	MSP_CHANNEL_TYPE	N/A	N/A	Type of MSPP channel. Can be one of the following values: • G711FullDuplex • G723FullDuplex • G726FullDuplex • G729FullDuplex • FaxRelayFullDuplex • G711EncodeSimplex • G711DecodeSimplex • G723EncodeSimplex • G723DecodeSimplex • G726EncodeSimplex • G726DecodeSimplex • G729EncodeSimplex • G729DecodeSimplex • RTPSwitching
FilterAttribs	WORD	Null	N/A	Enables or disables functionality in the channels. The following value is defined: MSP_FCN_ATTRIB_RFC2833: Enables inband DTMF carriage capability for the channel.

Channel parameter structures

Depending on the type of channel specified with the MSP_CHANNEL_TYPE parameter in the MSPP endpoint channel parameter structure, you must specify parameters in the following structures:

Channel type	Structure
Voice encode/decode (G.723.1, G.729A, G.726 simplex or duplex) or record channel	MSP_VOICE_CHANNEL_PARMS
T.38 fax	MSP_FAX_CHANNEL_PARMS
RTP switching	None. Only address parameters are required for switch channels.

This section discusses the Voice channel parameter structure and the T.38 fax relay channel parameter structure.

Voice channel parameter structures

When an application creates an MSPP API voice channel, it must specify the following parameters:

```
typedef struct tag_MSP_VOICE_CHANNEL_PARMS
{
    DWORD size;
    msp_FILTER_JITTER_PARMS JitterParms;
    msp_FILTER_ENCODER_PARMS EncoderParms;
    msp_FILTER_DECODER_PARMS DecoderParms;

} MSP_VOICE_CHANNEL_PARMS;
```

Record channels used in Dialogic native record functionality are voice channels that include only a jitter filter. Consequently, to specify initial native record channel settings, application use the MSP_VOICE_CHANNEL_PARMS structure, but specify only jitter parameters.

MSP_VOICE_CHANNEL_PARMS

Dependent function: mspCreateChannel

Field	Type	Default	Units	Description
size	DWORD	N/A	N/A	Size of the parameter structure.
msp_FILTER_JITTER_PARMS	Structure	N/A	N/A	Structure specifying parameters for jitter functionality.
msp_FILTER_ENCODER_PARMS	Structure	N/A	N/A	Structure specifying parameters for voice encoder functionality.
msp_FILTER_DECODER_PARMS	Structure	N/A	N/A	Structure specifying parameters for voice decoder functionality.

This topic discusses the following substructures:

- Jitter filter substructure
- Encoder filter substructure
- Decoder filter substructure

Jitter filter substructure

Applications specify jitter functionality with the following parameters:

```
typedef struct tag_msp_FILTER_JITTER_PARMS
{
    DWORD size;
    DWORD depth;
    DWORD adapt_enabled;
} msp_FILTER_JITTER_PARMS;
```

msp_FILTER_JITTER_PARMS

Dependent function: mspCreateChannel

Field name	Type	Default	Units	Description
size	DWORD	N/A	N/A	Size of the structure.
depth	DWORD	2	frames	Size (in frames) of the internal queue maintained by a jitter filter. Range is 1 to 16.
adapt_enabled	DWORD	0 (disabled)	1 =enabled 0 = disabled	Enables adaptive jitter on the channel jitter filter. For more information about adaptive jitter, refer to Jitter channel filter.

Encoder filter substructure

Applications specify voice encoder functionality with the following parameters:

```
typedef struct tag_msp_FILTER_ENCODER_PARMS
{
    DWORD size;
    WORD Mode;
    WORD Gain;
    WORD VadControl;
    WORD NotchControl;
    WORD IPFormat;
    WORD Rate;
    WORD PayloadID;
    WORD DtmfMode;
} msp_FILTER_ENCODER_PARMS;
```

msp_FILTER_ENCODER_PARMS

Dependent function: mspCreateChannel

Field name	Type	Default	Units	Description
size	DWORD	N/A	N/A	Size of the structure.
mode	WORD	0x0001 (online)	N/A	Vocoding mode. Off-line means that the encoder produces null media packet frames. 0x0000 = off-line 0x0001 = online
gain	WORD	0x0400 (0 dB)	N/A	Gain value applied to the encoder filter output in S5.10 format. Refer to Calculating vocoder filter gain and loss for a mapping of decibel gain/loss values to linear gain/loss values.
VadControl	WORD	0x0000 (VAD disabled)	N/A	Enables or disables voice activity detection (VAD). 0x0000 = VAD disabled 0x0001 = VAD enabled
NotchControl	WORD	0x0000 (DTMF/CED suppression disabled)	N/A	Enables and/or disables DTMF/CED tone suppression filters. The following values are defined: 0x0000 = DTMF and CED notch filters disabled 0x0001 = DTMF notch enabled, CED notch disabled 0x0002 = DTMF notch disabled, CED notch enabled 0x0003 = DTMF and CED notch filters enabled

Field name	Type	Default	Units	Description
IPFormat	WORD	0x0000	N/A	The companding algorithm applied to the IP payload (applies to G711 and G.726 encoders). The following values apply to G.711: • 0x0000 = mu-law (for T1 systems) • 0x0001 = A-law (for E1 systems) The following values apply to G.726 • 0x0000 = Little endian, RFC 3551 format • 0x0001 = Big endian, I.366.2 Annex E format
Rate	WORD	Encoder specific	N/A	Vocoding rate in kbit/s. Legal values depend on the encoder. Currently, only G.723.1 supports variable rates (where the default is 0, or 6.4 kbit/s). 0 specifies the highest bit rate setting. The bit rate decreases as settings are incremented by one. Therefore for G.723.1, 0 specifies 6.4 kbit/s and 1 specifies 5.3 kbit/s. If a 2 value is defined, it specifies a lower bit rate than 5.3 kbit/s.
PayloadID	WORD	Channel specific	N/A	Payload ID for outgoing Fusion media frames. Valid Range is 96 - 127. Refer to Vocoder filter attributes for a list of default payload IDs.

Field name	Type	Default	Units	Description
DtmfMode	WORD	DTMF_INBAND_ENABLED	N/A	Specifies the mode in which the filter processes DTMF tones. The application combines a payload value and a control value using the bitwise OR operator. The payload value is the DTMF payload ID shifted left by 8 bits. Defined values for control include: • DTMF_DISABLED Inband DTMF carriage is not supported for the channel. The following values can be ORed to specify encoder functionality: • DTMF_INBAND_ENABLED (default) MSPP channel detects DTMF tones and transfers DTMF information though RFC 2833 compliant packets. • DTMF_EVENTS_ENABLED MSPP channel detects DTMF tones and transfers DTMF information through MSPEVN_DPF_DTMF_REPORT events. • DTMF_SHIFT_ENABLED If set, shifts the timestamp of the associated DTMF RTP packets back to better align them with the actual start of DTMF detection. This allows for more synchronized decoder playout of the RFC 2833 DTMF digit with regard to actual DTMF time. • VOICE_ENABLED If DTMF_INBAND_ENABLED is set, the filter sends voice RTP packets as well as inband DTMF packets for the duration of the DTMF digit.

Decoder filter substructure

Applications specify voice decoder functionality with the following parameters:

```
typedef struct tag_msp_FILTER_DECODER_PARMS
{
    DWORD  size;
    WORD   Mode;
    WORD   Gain;
    WORD   IPFormat;
    WORD   PayloadID;
    WORD   DtmfMode;
} msp_FILTER_DECODER_PARMS;
```

msp_FILTER_DECODER_PARMS

Dependent function: mspCreateChannel

Field name	Type	Default	Units	Description
size	DWORD	N/A	N/A	Size of the structure.
mode	WORD	0x0001 (online)	N/A	Vocoding mode. Off-line means that the decoder filter plays silence to the DS0 endpoint. 0x0000= off-line 0x0001 = online
gain	WORD	0x0400 (0 dB)	N/A	Gain value applied to the decoder filter output in S5.10 format. Refer to Calculating vocoder filter gain and loss for a mapping of decibel gain/loss values to linear gain/loss values.
IPFormat	WORD	0x0000	N/A	The companding algorithm applied to the IP payload (applies to G.711 and G.726 decoders). The following values apply to G.711: • 0x0000 = mu-law (for T1 systems) • 0x0001 = A-law (for E1 systems) The following values apply to G.726: • 0x0000 = Little endian, RFC 3551 format • 0x0001 = Big endian, I.366.2 Annex E format

Field name	Type	Default	Units	Description
PayloadID	WORD	Channel specific	N/A	Payload ID for incoming Fusion media frames. Valid Range is 96 - 127. Refer to Vocoder filter attributes for a list of default payload IDs.
DtmfMode	WORD	DTMF_INBAND_ENABLED	N/A	Specifies if and how the decoder filter plays DTMF tones when it receives DTMF data (only when inband DTMF carriage is enabled). The application combines one or more control values with a playout value using the bitwise OR operator. Defined values for control include: • DTMF_DISABLED MSPP channel does not play a DTMF tone when it receives an RFC 2833 compliant inband DTMF carriage packet. • DTMF_INBAND_ENABLED MSPP channel automatically plays a DTMF tone when it detects an RFC 2833 compliant inband DTMF carriage packet. This setting is enabled by default. • DTMF_PLAY_ENABLED MSPP channel plays DTMF tones when the application sends an MSP_CMD_DECODE_PLAY_DTMF_DIGIT command. The playout value is an integer that specifies the number of decoder frames for an ongoing tone to continue when no end-of-tone packet is received. The integer must be shifted left by 8 bits before being ORed with one or more control values.

T38 fax relay channel parameter structures

When an application creates an MSPP API T38UFP fax channel, it must specify the following parameters:

```
typedef struct tag_MSP_FAX_CHANNEL_PARMS
{
    DWORD size;
    msp_FILTER_FAXRELAY_CONFIG FaxRelayParms;

} MSP_FAX_CHANNEL_PARMS;
```

MSP_FAX_CHANNEL_PARMS

Dependent function: mspCreateChannel

Field	Type	Default	Units	Description
size	DWORD	N/A	N/A	Size of the parameter structure.
msp_FILTER_FAXRELAY_CONFIG	Structure	N/A	N/A	Structure specifying parameters for fax modem functionality.

T.38 fax relay configuration channel parameter structure

When an application creates an MSPP API T.38 fax channel, it must specify the following parameters:

```
typedef struct tag_msp_FILTER_FAXRELAY_CONFIG
{
    WORD    modemask;
    WORD    eventmask;
    WORD    maxbitrate;
    WORD    timeout;

 } msp_FILTER_FAXRELAY_CONFIG;
```

msp_FILTER_FAXRELAY_CONFIG

Dependent function: mspCreateChannel

Field	Type	Default	Units	Description
modemask	WORD	NSF enabled (ECM disabled)	N/A	Enables and disables specific T.38 fax features. The following values are defined: FAXRELAY_MODEMASK_DEFAULT FAXRELAY_MODEMASK_DISABLE_NSF_FILTERING FAXRELAY_MODEMASK_DISABLE_ECM FAXRELAY_MODEMASK_ENABLE_NON_STD_T38_ECM

Field	Type	Default	Units	Description
eventmask	WORD	All enabled	N/A	Specifies the unsolicited events that are passed back to the application. The following values are defined: FAXRELAY_EVENTMASK_DEFAULT Reports no fax session information. FAXRELAY_EVENTMASK_REPT_FAX_BEGIN_END Reports the beginning and ending of a fax session. FAXRELAY_EVENTMASK_REPT_PASSED_PAGE Reports successful passing of a fax page.
maxbitrate	WORD	14400	Bits/s	Maximum bit rate of the fax transmission (in bits/sec). FAXRELAY_MAXBITRATE_DEFAULT FAXRELAY_MAXBITRATE_14400 FAXRELAY_MAXBITRATE_9600 FAXRELAY_MAXBITRATE_4800 FAXRELAY_MAXBITRATE_2400 FAXRELAY_MAXBITRATE_DISABLE_FILTER
timeout	WORD	15000 ms.	ms	Fax session timeout value. The following constants are defined for this parameter: FAXRELAY_TIMEOUT_DEFAULT FAXRELAY_TIMEOUT_MIN (0 ms.) FAXRELAY_TIMEOUT_MAX (65000)

Note: Switching channels have no associated parameters.

12. Endpoint parameters

Overview of endpoint structures

When creating MSPP endpoints with **mspCreateEndpoint**, you must specify endpoint configuration information in two structures:

- MSPP address structure
- MSPP parameter structure

MSPP address structures

Depending on the type of endpoint you are creating, specify the following parameters in the MSP_ENDPOINT_ADDR structure:

```
typedef struct tag_MSP_ENDPOINT_ADDR
{
    DWORD   size;
    DWORD   nBoard;
    DWORD   eEpType;

    union
    {
        DS0_ENDPOINT_ADDR           DS0;
        RTPRTCP_ENDPOINT_ADDR       RtpRtcp;
        RTPRTCP_V6_ENDPOINT_ADDR    RtpRtcpV6;
        T38UDP_ENDPOINT_ADDR        T38Udp;

        // Structure may be expanded to define new Endpoints
    } EP;

} MSP_ENDPOINT_ADDR;
```

Depending on the value specified in the eEpType field, the application also specifies other endpoint address information.

MSP_ENDPOINT_ADDR

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
size	DWORD	N/A		
nBoard	DWORD	N/A	N/A	Endpoint location (board number).

Field	Type	Default	Units	Description
eEpType	DWORD	N/A	N/A	Type of MSPP endpoint. Can be the following: MSP_ENDPOINT_DS0 MSP_ENDPOINT_RTPIN MSP_ENDPOINT_RTPOUT MSP_ENDPOINT_RTPFDX MSP_ENDPOINT_RTPIN_V6 MSP_ENDPOINT_RTPOUT_V6 MSP_ENDPOINT_RTPFDX_V6 MSP_ENDPOINT_T38UDP MSP_ENDPOINT_TPKT

For information about endpoint address structures, see:

- DS0 endpoint address structure.
- RTP IPv4 endpoint address structure.
- RTP IPv6 endpoint address structure.
- T38UDP endpoint address structure.
- TPKT endpoint address structure. (No address structure is required for TPKT endpoints.)

MSPP parameter structures

Depending on the type of endpoint you are creating, specify the following parameters in the MSP_ENDPOINT_PARAMETER structure:

```
typedef struct tag_MSP_ENDPOINT_PARMS
{
    DWORD   size;
    DWORD   eParmType;

    union
    {
        DS0_ENDPOINT_PARMS        DS0;
        RTPRTCP_ENDPOINT_PARMS    RtpRtcp;
        RTPRTCP_V6_ENDPOINT_PARMS RtpRtcpV6;
        T38UDP_ENDPOINT_PARMS     T38Udp;
        TPKT_ENDPOINT_PARMS       Tpkt;
    } EP;

} MSP_ENDPOINT_PARAMETER;
```

MSP_ENDPOINT_PARAMETER

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
size	DWORD	N/A	N/A	Size of the parameter structure.

Field	Type	Default	Units	Description
eParmType	DWORD	N/A	N/A	Type of MSPP endpoint. Can be one of the following: MSP_ENDPOINT_DS0 MSP_ENDPOINT_RTPIN MSP_ENDPOINT_RTPOUT MSP_ENDPOINT_RTPFDX MSP_ENDPOINT_RTPIN_V6 MSP_ENDPOINT_RTPOUT_V6 MSP_ENDPOINT_RTPFDX_V6 MSP_ENDPOINT_T38UDP MSP_ENDPOINT_TPKT

For information about endpoint parameter structures, see:

- DS0 endpoint parameter structure.
- RTP IPv4 endpoint parameter structure.
- RTP IPv6 endpoint parameter structure.
- T38UDP endpoint parameter structure.
- TPKT endpoint parameter structure.

DS0 endpoint address structure

When the endpoint type is MSP_ENDPOINT_DS0, the application provides the following endpoint address information:

```
typedef struct tag_DS0_ENDPOINT_ADDR
{
    DWORD   nTimeslot;
} DS0_ENDPOINT_ADDR;
```

DS0_ENDPOINT_ADDR

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
nTimeslot	DWORD	N/A	N/A	CT bus timeslot address.

DS0 endpoint parameter structure

When the endpoint type is MSP_ENDPOINT_DS0, the application provides the following parameters:

```
typedef struct tag_DS0_ENDPOINT_PARMS
{

    DWORD     size;
    MSP_MEDIA media;

} DS0_ENDPOINT_PARMS;
```

DS0_ENDPOINT_PARMS

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
media	MSP_MEDIA	N/A	N/A	Type of data to be transferred through the endpoint. Must be one of the following: MSP_VOICE MSP_FAX

RTP IPv4 endpoint address structure

When the endpoint type is MSP_ENDPOINT_RTPFDX, MSP_ENDPOINT_RTPIN, or MSP_ENDPOINT_RTPOUT, the application provides the following endpoint address information:

```
typedef struct tag_RTPRTCP_ENDPOINT_ADDR
{
    char   DestIpAddress[20];
    WORD   nDestPort;
    char   SrcIpAddress[20];
    WORD   nSrcPort;

} RTPRTCP_ENDPOINT_ADDR;
```

RTPRTCP_ENDPOINT_ADDR

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
DestIpAddress[20]	IPADDRESS	N/A	N/A	Destination IPv4 address in dot notation. NULL for RTP simplex receive (MSP_ENDPOINT_RTPIN) endpoints.
nDestPort	WORD	N/A	N/A	Destination port number. NULL for RTP simplex receive (MSP_ENDPOINT_RTPIN) endpoints.
SrcIpAddress[20]	IPADDRESS	N/A	N/A	Source IPv4 address in dot notation. RTP simplex send (MSP_ENDPOINT_RTPOUT) endpoints.
nSrcPort	WORD	N/A	N/A	Source port number. NULL for RTP simplex SEND (MSP_ENDPOINT_RTPOUT) endpoints.

RTP IPv4 endpoint parameter structure

When the endpoint type is MSP_ENDPOINT_RTPFDX, MSP_ENDPOINT_RTPIN, or MSP_ENDPOINT_RTPOUT, the application provides the following endpoint parameters:

```
typedef struct tag_RTPRTCP_ENDPOINT_PARMS
{

   DWORD   size;
   BYTE    TypeOfService;
   DWORD   startRtcp;
   RTCP_SESSION_PARMS rtcpParms;
   DWORD   RtpTsFreq;
   DWORD   Session_bw;
   DWORD   dtmf_event_control;
   DWORD   frameQuota;
   DWORD   linkEvents;
  RTP_ PAYLOAD_MAP   PayloadMap;

} RTPRTCP_ENDPOINT_PARMS;
```

RTPRTCP_ENDPOINT_PARMS

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
TypeOfService	BYTE	0	integer	IPv4 ToS Type of Service field in IP header.
startRcp	DWORD	0	integer	Set to non-zero to start an RTCP session. Specify the value of the ToS field for RTCP packets in the upper 16 bits of this field, or by using a macro: startRtcp = 1 | MSPP_SET_RTCP_TOS (ToS). RTCP_SESSION_PARMS parameters must be specified for active RTCP sessions.
RtpTsFreq	DWORD	8000	Hz	Timestamp frequency.
Session_bw	DWORD	64000	Hz	Session bandwidth (for future use only).

Field	Type	Default	Units	Description
rfc2833_event_control	DWORD	SEND_FIRST_EVENT and SEND_LAST_EVENT	N/A	Combination of two 16 bit WORDs that determine how the endpoint responds when receiving an RFC 2833 compliant packet payload. A control value specifies how the endpoint responds to RFC 2833 compliant packets. Defined control values include (control values can be ORed together): SEND_NO_EVENTS No MSPEVN_RFC2833_REPORT event is sent upon arrival of an RFC 2833 compliant packet. SEND_FIRST_EVENT MSPEVN_RFC2833_REPORT event is sent upon the arrival of the first RFC 2833 compliant packet for a given DTMF digit. SEND_LAST_EVENT MSPEVN_RFC2833_REPORT event is sent upon the arrival of the last RFC 2833 compliant packet for a given DTMF digit. SEND_ALL_EVENTS MSPEVN_RFC2833_REPORT event is sent upon arrival of all RFC 2833 compliant packets and is limited by the decimation value. A decimation value specifies how many RFC 2833 compliant packets the filter should receive for a particular tone before sending another event.
frameQuota	DWORD	2	Number of frames	Number of media frames assembled in each packet payload.
linkEvents	DWORD	0	1=enabled 0=disabled	Whether the MSPP API returns route availability events that report CG board Ethernet interface status transitions.

Field	Type	Default	Units	Description
PayloadMap	Structure	N/A	N/A	Structure for specifying payload ID map for the endpoint. The structure consists of the following: `typedef struct tag_RTP_PAYLOAD_MAP { DWORD vocoder; DWORD payload_id; } RTP_PAYLOAD_MAP;` Where vocoder specifies the string of a supported Fusion vocoder, and payload_id specifies a payload ID number to specify for this vocoder. For more information about vocoder identification strings and default payload IDs for supported vocoders, refer to the *mspcmd.h* header file.

RTCP session parameter substructure

If the application enables RTCP reports through the startrcp parameter in the RTPRTCP_ENDPOINT_PARMS structure, it must also include the following substructure:

```
typedef struct TAG_RTCP_SESSION_PARMS
{
    DWORD       forwardPkts;
    char        cname[32];
    char        name[32];
    char        email[32];
    char        phone[32];
    char        location[32];
    char        tool[32];
    char        note[32];
} RTCP_SESSION_PARMS;
```

RTCP_SESSION_PARMS

Dependent function: mspCreateEndpoint

Field name	Type	Default	Units	Description
forwardPkts	DWORD	N/A	1, <>1	If set to true (1), all RTCP packets are forwarded to the application.
cname	char	N/A	N/A	NULL terminated string containing the canonical name identifier.
name	char	N/A	N/A	NULL terminated string containing the user name.

Field name	Type	Default	Units	Description
email	char	N/A	N/A	NULL terminated string containing the email address.
phone	char	N/A	N/A	NULL terminated string containing the phone number.
location	char	N/A	N/A	NULL terminated string containing the geographic user location.
tool	char	N/A	N/A	NULL terminated string containing the application or tool name.
note	char	N/A	N/A	NULL terminated string containing the notice/status.

RTP IPv6 endpoint address structure

When the endpoint type is MSP_ENDPOINT_RTPFDX_V6, MSP_ENDPOINT_RTPIN_V6, or MSP_ENDPOINT_RTPOUT_V6, the application provides the following endpoint address information:

```
typedef struct tag_RTPRTCP_V6_ENDPOINT_ADDR
{
    char   DestIpv6Address[46];
    WORD   nDestPort;
    WORD   ifc_idx;
    char   SrcIpv6Address[46];
    WORD   nSrcPort;

} RTPRTCP_V6_ENDPOINT_ADDR;
```

RTPRTCP_ENDPOINT_ADDR

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
DestIPv6Address [46]	IPADDRESS	N/A	N/A	Destination IPv6 address. NULL for IPv6 RTP simplex receive (MSP_ENDPOINT_RTPIN_V6)endpoints.
nDestPort	WORD	N/A	N/A	Destination port number. NULL for IPv6 RTP simplex receive (MSP_ENDPOINT_RTPIN_V6) endpoints.
ifc_idx	WORD	N/A	N/A	CG board Ethernet interface number (1 or 2). Specifying 0 allows the IPv6 stack to choose.

Field	Type	Default	Units	Description
SrcIPv6Address[46]	IPADDRESS	N/A	N/A	Source IPv6 address. NULL for IPv6 RTP simplex send (MSP_ENDPOINT_RTPOUT_V6) endpoints.
nSrcPort	WORD	N/A	N/A	Source port number. NULL for IPv6 RTP simplex SEND (MSP_ENDPOINT_RTPOUT_V6) endpoints.

RTP IPv6 endpoint parameter structure

When the endpoint type is MSP_ENDPOINT_RTPFDX_V6, MSP_ENDPOINT_RTPIN_V6, or MSP_ENDPOINT_RTPOUT_V6, the application provides the following endpoint parameters:

```
typedef struct tag_RTPRTCP_V6_ENDPOINT_PARMS
{

   DWORD   size;
   BYTE    trafficClass
   DWORD   flowLabel
   DWORD   startRcp;
RTCP_SESSION_PARMS rtcpParms;
   DWORD   RtpTsFreq;
   DWORD   Session_bw;
   DWORD   dtmf_event_control;
   DWORD   frameQuota;
   DWORD   linkEvents;
   RTP_ PAYLOAD_MAP   PayloadMap;

} RTPRTCP_V6_ENDPOINT_PARMS;
```

RTPRTCP_ENDPOINT_PARMS

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
trafficClass	BYTE	0	N/A	Class or priority of the IPv6 packet.
flowLabel	DWORD	0	N/A	Specific sequence to which the IPv6 packet belongs.
startRcp	DWORD	0	integer	Set this to non-zero to start RTCP session. RTCP_SESSION_PARMS parameters must be specified for active RTCP sessions.
rtcpParms	Structure	N/A	N/A	Structure specifying RTCP session parameters.
RtpTsFreq	DWORD	8000	Hz	Timestamp frequency.

Field	Type	Default	Units	Description
Session_bw	DWORD	64000	Hz	Session bandwidth (for future use only).
dtmf_ event_ control	DWORD	SEND_ FIRST_ EVENT and SEND_ LAST_ EVENT	N/A	Combination of two 16 bit WORDs that determine how the endpoint responds when receiving an RFC 2833 compliant packet payload. A control value specifies how the endpoint responds to RFC 2833 compliant packets. Defined control values include (control values can be ORed together): SEND_NO_EVENTS No MSPEVN_RFC2833_REPORT event is sent upon arrival of an RFC 2833 compliant packet. SEND_FIRST_EVENT MSPEVN_RFC2833_REPORT event is sent upon the arrival of the first RFC 2833 compliant packet for a given DTMF digit. SEND_LAST_EVENT MSPEVN_RFC2833_REPORT event is sent upon the arrival of the last RFC 2833 compliant packet for a given DTMF digit. SEND_ALL_EVENTS MSPEVN_RFC2833_REPORT event is sent upon arrival of all RFC 2833 compliant packets and is limited by the decimation value. A decimation value specifies how many RFC 2833 compliant packets the filter should receive for a particular tone before sending another event.
frameQuota	DWORD	2	Number of frames	Number of media frames assembled in each packet payload.
linkEvents	DWORD	0	1=enabled 0=disabled	Whether the MSPP API returns route availability events that report CG board Ethernet interface status transitions.

Field	Type	Default	Units	Description
PayloadMap	Structure	N/A	N/A	Structure for specifying payload ID map for the endpoint. The structure consists of the following: `typedef struct tag_RTP_PAYLOAD_MAP { DWORD vocoder; DWORD payload_id; } RTP_PAYLOAD_MAP;` Where vocoder specifies the string of a supported Fusion vocoder, and payload_id specifies a payload ID number to specify for this vocoder. For more information about vocoder identification strings and default payload IDs for supported vocoders, refer to the *mspcmd.h* header file.

RTCP session parameter substructure

If the application enables RTCP reports through the startrcp parameter in the RTPRTCP_ENDPOINT_PARMS structure, it must also include the following substructure:

```
typedef struct TAG_RTCP_SESSION_PARMS
{
    DWORD       forwardPkts;
    char        cname[32];
    char        name[32];
    char        email[32];
    char        phone[32];
    char        location[32];
    char        tool[32];
    char        note[32];
} RTCP_SESSION_PARMS;
```

RTCP_SESSION_PARMS

Dependent function: mspCreateEndpoint

Field name	Type	Default	Units	Description
forwardPkts	DWORD	N/A	1, <>1	If set to true (1), all RTCP packets are forwarded to the application.
cname	char	N/A	N/A	NULL terminated string containing the canonical name identifier.
name	char	N/A	N/A	NULL terminated string containing the user name.
email	char	N/A	N/A	NULL terminated string containing the email address.

Field name	Type	Default	Units	Description
phone	char	N/A	N/A	NULL terminated string containing the phone number.
location	char	N/A	N/A	NULL terminated string containing the geographic user location.
tool	char	N/A	N/A	NULL terminated string containing the application or tool name.
note	char	N/A	N/A	NULL terminated string containing the notice/status.

T38UDP endpoint address structure

When the endpoint type is MSP_ENDPOINT_T38FDX, the application provides the following endpoint address information:

```
typedef struct tag_T38UDP_ENDPOINT_ADDR
{
    // UDP parameters
    char   DestIpAddress[20];
    WORD   nDestPort;
    char   SourceIpAddress[20];
    WORD   nSourcePort;

} T38UDP_ENDPOINT_ADDR;
```

T38UDP_ENDPOINT_ADDR

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
pDestIpAddress[20]	char	N/A	N/A	Destination IPv4 address in dot notation.
nDestPort	WORD	N/A	N/A	Destination UDP port number.
pSourceIpAddress[20]	char	N/A	N/A	Source IPv4 address in dot notation.
nSourcePort	WORD	N/A	N/A	Source UDP port number.

T38UDP endpoint parameter structure

When the endpoint type is MSP_ENDPOINT_T38FDX, the application provides the following parameters:

```
typedef struct  tag_T38UDP_ENDPOINT_PARMS
{
    DWORD                   size;
    BYTE                    TypeOfService;
    msp_T38UDPTLDEC_CONFIG  UdptlDecodeParms;
    msp_T38UDPTLENC_CONFIG  UdptlEncodeParms;
    msp_T38IFPDEC_CONFIG    IfpDecodeParms;
    msp_T38IFPENC_CONFIG    IfpEncodeParms;

} T38UDP_ENDPOINT_PARMS;
```

This topic discusses the following substructures:

- T38UDP endpoint IFP decoder substructure
- T38UDP endpoint IFP encoder substructure
- T38UDP endpoint UDPTL decoder substructure
- T38UDP endpoint UDPTL encoder substructure

T38UDP_ENDPOINT_PARMS

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
size	DWORD	N/A	N/A	Size of the parameter structure.
typeofservice	Byte	N/A	N/A	IPv4 type of service field in IP header.
msp_T38UDPTLDEC_CONFIG	Structure	N/A	N/A	Structure specifying parameters for UDPTL decoder functionality.
msp_T38UDPTLENC_CONFIG	Structure	N/A	N/A	Structure specifying parameters for UDPTL encoder functionality.
msp_T38IFPDEC_CONFIG	Structure	N/A	N/A	Structure specifying parameters for IFP decoder functionality.
msp_T38IFPENC_CONFIG	Structure	N/A	N/A	Structure specifying parameters for IFP encoder functionality.

T38UDP endpoint IFP decoder substructure

Applications specify T.38 fax IFP decoder functionality with the following parameters:

```
typedef struct tag_msp_T38IFPDEC_CONFIG
{
    DWORD   queue_size;
    DWORD   buffering_time;
    DWORD   page_data_keep_alive_time;
} msp_T38IFPDEC_CONFIG;
```

msp_T38IFPDEC_CONFIG

Dependent function: mspCreateEndpoint

Field name	Type	Default	Units	Description
queue_size	DWORD	20	IFP packets	Size (in number of IFP packets) of the reorder queue used for storing and reordering incoming packets and performing error recovery. Defined values include: T38IFPDEC_QUEUE_SIZE_ DEFAULT (20 packets) T38IFPDEC_QUEUE_SIZE_MIN (10 packets) T38IFPDEC_QUEUE_SIZE_MAX (100 packets)
buffering_time	DWORD	200	ms.	Changes the buffering time that elapses before sending data to a T.38 fax modulator/demodulator filter, in order to pre- fill the reordering queue. Time is specified in milliseconds and should be in multiples of 10. Defined values include: T38IFPDEC_BUFFERING_ TIME_DEFAULT (200 ms) T38IFPDEC_BUFFERING_ TIME_MIN (0 ms) T38IFPDEC_BUFFERING_ TIME_MAX (1500 ms)

Field name	Type	Default	Units	Description
page_data_keep_alive_time	DWORD	0	ms.	Sets the amount of time to keep a fax session in the page transmission phase of a fax call in the absence of data from the packet network (that is, absence of data in the IFP reorder queue). Defined values include: T38IFPDEC_PAGE_DATA_KEEP_ALIVE_TIME_DEFAULT (0 ms) T38IFPDEC_PAGE_DATA_KEEP_ALIVE_TIME_MIN (0 ms) T38IFPDEC_PAGE_DATA_KEEP_ALIVE_TIME_MAX (4000 ms)

T38UDP endpoint IFP encoder substructure

Applications specify T.38 fax IFP decoder functionality in the following parameters:

```
typedef struct tag_msp_T38IFPENC_CONFIG
{
    DWORD   v21_bytes_per_ifp_packet;
    DWORD   page_modem_ifp_duration;
    DWORD   t30_indicator_send_count;
    DWORD   t30_indicator_send_period;
} msp_T38IFPENC_CONFIG;
```

msp_T38IFPENC_CONFIG

Dependent function: mspCreateEndpoint

Field name	Type	Default	Units	Description
v21_bytes_per_ifp_packet	DWORD	1	Number of V21 bytes	Number of V.21 bytes per T.38 IFP packet. Range is 1-128. Defined values include: T38IFPENC_V21_BYTES_PER_PACKET_DEFAULT (1 byte) T38IFPENC_V21_BYTES_PER_PACKET_MIN (1 byte) T38IFPENC_V21_BYTES_PER_PACKET_MAX (128 bytes)

Field name	Type	Default	Units	Description
page_modem_ifp_duration	DWORD	30	ms.	Number of 10 ms. page modem frames per IFP packet. Range is 10 - 100. Values must be specified in 10 ms increments. Defined values include: T38IFPENC_PAGE_MODEM_IFP_DURATION_DEFAULT (30 ms) T38IFPENC_PAGE_MODEM_IFP_DURATION_MIN (10 ms) T38IFPENC_PAGE_MODEM_IFP_DURATION_MAX (100 ms)
t30_indicator_send_count	DWORD	3	packets	Number of T.38 indicator packets to transmit to the network. Range is 1 - 65000. Defined values include: T38IFPENC_T30_INDICATOR_SEND_COUNT_DEFAULT (3 packets) T38IFPENC_T30_INDICATOR_SEND_COUNT_MIN (1 packet) T38IFPENC_T30_INDICATOR_SEND_COUNT_MAX (65000 packets)
t30_indicator_send_period	DWORD	10	ms.	Specifies the interval between T.38 indicators transmitted to the network. Must be specified in 10 ms. increments. Defined values include: T38IFPENC_T30_INDICATOR_SEND_PERIOD_DEFAULT (10 ms) T38IFPENC_T30_INDICATOR_SEND_PERIOD_MIN (10 ms) T38IFPENC_T30_INDICATOR_SEND_PERIOD_MAX (1000 ms)

T38UDP endpoint UDPTL decoder substructure

Applications specify T.38 fax IFP decoder functionality in the following parameter:

```
typedef struct tag_msp_T38UDPTLDEC_CONFIG
{
    DWORD   enable_error_recovery;
} msp_T38UDPTLDEC_CONFIG;
```

msp_T38UDPTLDEC_CONFIG

Dependent function: mspCreateEndpoint

Field name	Type	Default	Units	Description
enable_error_recovery	Boolean	1	N/A	Enables or disables T.38 fax error decoding and recovery. Allowed values include 1 (enabled) and 0 (disabled).

T38UDP endpoint UDPTL encoder substructure

Applications specify T.38 fax IFP encoder functionality in the following parameters:

```
typedef struct tag_msp_T38UDPTLENC_CONFIG
{
    DWORD   error_protection_scheme;
    union
    {
        T38UDPTL_REDUNDANCY_CFG  redundancy;
        T38UDPTL_PARITY_FEC_CFG  parity_fec; /NOT SUPPORTED/
    } eps;
   DWORD   num_duplicate_v21_udptl_packets;
   DWORD   num_duplicate_indicator_udptl_packets;
   DWORD   num_duplicate_page_modem_udptl_packets;
} msp_T38UDPTLENC_CONFIG;
```

msp_T38UDPTLENC_CONFIG

Dependent function: mspCreateEndpoint

Field name	Type	Default	Units	Description
error_ protection_ scheme	DWORD	Redundancy error protection scheme	N/A	Specifies the Error protection scheme used by the filter. Defined values include: T38UDPTL_EPS_DEFAULT Redundancy error protection scheme. If the application want to enable redundancy error protection, it provides the a T38UDPTL_EPS_REDUNDANCY structure which a single parameter: redundant_ifp_packets Defined values for this parameter include: T38UDPTLENC_ REDUNDANT_IFPS_DEFAULT (3 IFP packets per UDPTL) T38UDPTLENC_ REDUNDANT_IFPS_MIN (0 IFP packets per UDPTL) T38UDPTLENC_ REDUNDANT_IFPS_DEFAULT (4 IFP packets per UDPTL)
num_duplicate_ v21_udptl_ packets	DWORD	0	packets	Number of duplicate V.21 UDPTL packets to transmit. Defined values include: T38UDPTLENC_PACKET_ DUPLICATION_V21_DEFAULT (0 packets) T38UDPTLENC_PACKET_ DUPLICATION_V21_MIN (0 packets) T38UDPTLENC_PACKET_ DUPLICATION_V21_MAX (5 packets)

Field name	Type	Default	Units	Description
num_duplicate_ indicator _udptl_packets	DWORD	0	packets	Number of duplicate indicator UDPTL packets to transmit. Defined values include: T38UDPTLENC_PACKET_ DUPLICATION_INDICATOR_ DEFAULT (0 packets) T38UDPTLENC_PACKET_ DUPLICATION_INDICATOR_MIN (0 packets) T38UDPTLENC_PACKET_ DUPLICATION_INDICATOR_MAX (5 packets)
num_duplicate_ page_modem _udptl_packets	DWORD	0	packets	Number of duplicate page modem UDPTL packets to transmit. Defined values include: T38UDPTLENC_PACKET_ DUPLICATION_PAGE_MODEM_ DEFAULT (0 packets) T38UDPTLENC_PACKET_ DUPLICATION_PAGE_MODEM_ MIN (0 packets) T38UDPTLENC_PACKET_ DUPLICATION_PAGE_MODEM_ MAX (5 packets)

TPKT endpoint address structure

Unlike other MSPP endpoint types, ThroughPacket endpoints do not require an address structure. When applications create TPKT endpoints with **mspCreateEndpoint**, they pass a NULL value for the endpoint address value.

TPKT endpoint parameter structure

When the endpoint type is MSP_ENDPOINT_TPKT, the application provides the following parameters:

```
typedef struct tag_TPKT_ENDPOINT_PARMS
{
    DWORD   localSessionID;
    DWORD   localSessionSeq;
    DWORD   remoteSessionID;
    DWORD   remoteSessionSeq;
    DWORD   deliveryMethod;
    BYTE    remoteGatewayIP[4];
    DWORD   coderType;

} TPKT_ENDPOINT_PARMS;
```

TPKT_ENDPOINT_PARMS

Dependent function: mspCreateEndpoint

Field	Type	Default	Units	Description
localSessionID	DWORD	N/A	N/A	Local session ID. Range is 0 to 4094.
localSessionSeq	DWORD	N/A	N/A	Sequence number for detecting session overlap. Can be set to 0 or 1.
remoteSessionID	DWORD	N/A	N/A	Remote session ID. Range is 0 to 4094.
remoteSessionSeq	DWORD	N/A	N/A	Sequence number for detecting session overlap. Can be set to 0 or 1.
deliveryMethod	DWORD	N/A	N/A	Session delivery method (simple or complex): ComplexPacket: complex packets SimplePacket: simple packets
remoteGatewayIP	BYTE	N/A	N/A	Destination gateway IPv4 address for the endpoint.
coderType	DWORD	N/A	N/A	TPKT endpoint vocoder type. Valid coder types include: G723High: G.723.1 6.4 kbit/s G723Low: G.723.1 5.3 kbit/s G729: G.729 (A) at 8.0 kbit/s G711Alaw_64: G.711 A-law at 64 kbit/s G711Mulaw: G.711 mu-law at 64 kbit/s G726_32: G.726 ADPCM at 32 kbit/s

For more information about creating and configuring ThroughPacket endpoints, refer to the *Dialogic® NaturalAccess™ Fusion VoIP API Developer's Manual*.

13. MSPP voice encoder and decoder attributes

Vocoder filter attributes

The following table shows MSPP encoder/decoder filter attributes:

Vocoder	Literal (RFC 1890 compliant)	Data rate (kbit/s)	VAD	Sample rate	Static payload ID (RFC 1890 compliant)	Dialogic default payload ID	Media packet latency	Payload size (bytes)	RTP timestamp increments
mu-law PCM	PCMU	64	Yes	8000	0	0	10	80	80
G.726 (at 32 kbit/s)	G726_2	32	No	8000	2	2	10	40	80
G.723.1	G723	6.4/5.3 (default=6.4)	Yes	8000	4	4	30	24, 20 (4)	240
A-law PCM	PCMA	64	Yes	8000	8	8	10	80	80
G.729A	G729	8.0	Yes	8000	18	18	10	10 (2)	80

Calculating vocoder filter gain and loss

The following table shows decibel gain values and their associated linear gain/loss values:

Decibel gain/loss	Gain	Loss
0 dB	0x0400	NA
1 dB	0x047D	0x0391
2 dB	0x0509	0x032D
3 dB	0x05A6	0x02D5
4 dB	0x0657	0x0286
5 dB	0x071D	0x0240
6 dB	0x07FB	0x0201
7 dB	0x08F4	0x01C9

Decibel gain / loss	Gain	Loss
8 dB	0x0A0C	0x0198
9 dB	0x0B46	0x016B
10 dB	0x0CA6	0x0144
11 dB	0x0E31	0x0121
12 dB	0x0FED	0x0101
13 dB	NA	0x00E5
14 dB	NA	0x00CC
15 dB	NA	0x00B6
16 dB	NA	0x00A2
17 dB	NA	0x0091
18 dB	NA	0x0081
squelch	NA	0x0000

Converting from decibels to API logarithmic units

Use the following formula to convert values from dB to API logarithmic units (where DSP 0 dB ref is equivalent to 1024 (0x0400)):

```
API Units = ( DSP 0 dB ref ) * (10^(dB/20))
For example, to compute a 3dB gain value:
   3dB API Units = 1024 * 10^(3/20)
   1024 * 10^(.15)
   1024 * 1.4125
   1446
   0x05A6
```

14. Index