endace
accelerated

Data Stream Manager API Programming Guide

EDM04-10

www.endace.com

Published by:

Endace Limited

Building 7

Lambie Drive

PO Box 76802
Manukau City 1702
New Zealand

Phone: +64 9 262 7260

Fax: +64 9 262 7261

support@endace.com

www.endace.com

International Locations

New Zealand	Americas	Europe, Middle East & Africa
Endace Technology® Ltd	Endace USA® Ltd	Endace Europe® Ltd
Level 9 85 Alexandra Street	Suite 220 11495 Sunset Hill Road	Sheraton House Castle Park
PO Box 19246 Hamilton 2001 New Zealand	Reston Virginia 20190 United States of America	Cambridge CB3 0AX United Kingdom
Phone: +64 7 839 0540 Fax: +64 7 839 0543	Phone: +1 703 382 0155 Fax: +1 703 382 0155	Phone: +44 1223 370 176 Fax: +44 1223 370 040

Table of Contents

Chapter 1: Introduction

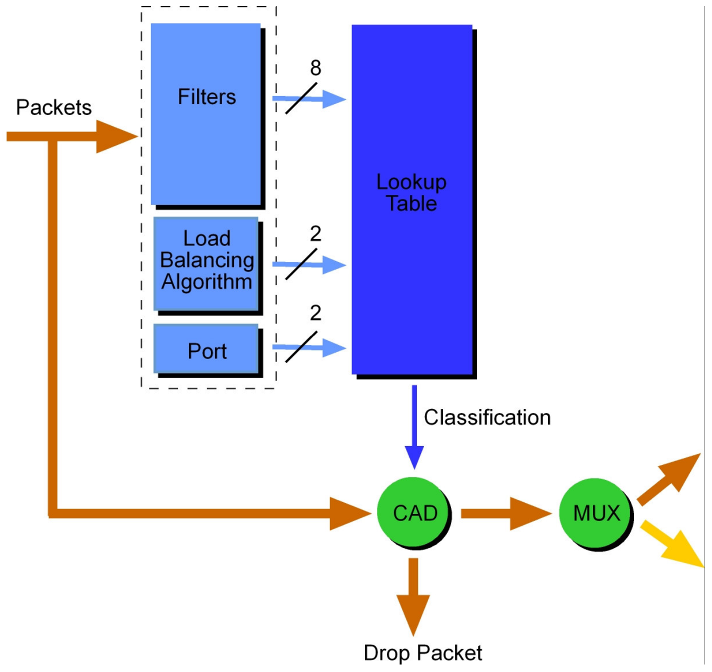
Overview

The Data Stream Manager (DSM) is a feature supported on the following DAG Cards:

- DAG 4.5G2/G4/GF
- DAG 6.2S
- DAG 5.2X
- DAG 8.2X

DSM allows you to drop or route packets to a particular receive stream based on the packet contents, physical port and the output of two load balancing algorithms. The DSM logic is implemented in firmware on the DAG card, it does not require intervention from the host CPU once it is configured.

The diagram below shows the logical flow of packet records in the DSM module:



Packets are received from the line and stamped with an ERF (Endace Record Format) header, then past along to the filter and load balancing block.

- **Filter/Load Balancing Block**
  The filter block applies eight bit-mask filters simultaneously to the start of the packet, producing a single true/false value for each filter. The load balancing block applies two algorithms to the packet data, again producing one true/false Boolean output per algorithm.

  Note: Load Balancing (LB) is also known as Hash Load Balancing (HLB) or simply as steering algorithm

- **Lookup Table Block**
  The lookup table accepts the filter and load balancing outputs. It also receives the physical port the packet arrived on and calculates a classification (known as the color) for the packet.

Overview (cont.)

- **Coloriser and Drop Block**
  The color is then past onto the Coloriser And Drop (CAD) block that checks if the packet should be dropped, if not the color is inserted into the ERF record header of the packet and then the packet record is past along to the packet record multiplexer.
- **Packet Record Multiplexer (ERF MUX)**
  The ERF MUX looks at the color information contained in the packet record and determines to which receive stream the packet record should be routed to.

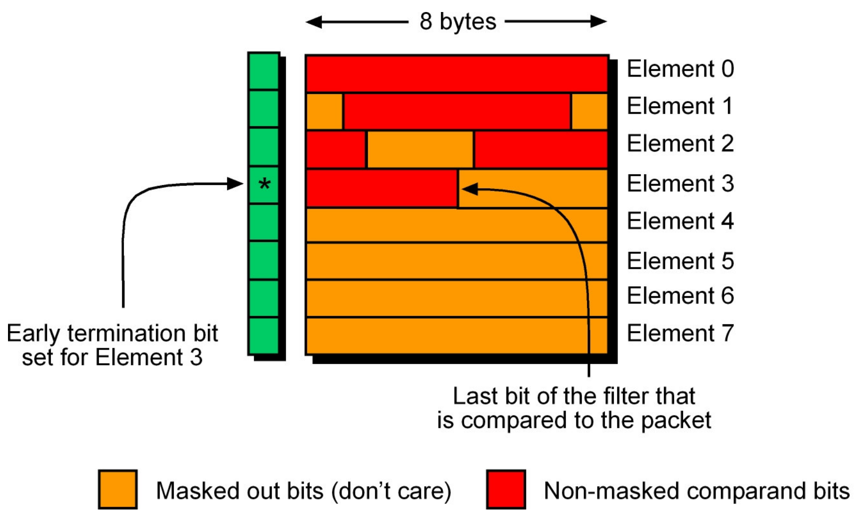
Packet Filters

Prior to packets being presented to the DSM module, they are stamped with a ERF record header and possibly snapped to a particular length (set by the `snap length` card configuration option). This is standard DAG card behaviour, but should be taken into account when using the DSM firmware as it could effect filter output.

There are eight 64-byte bitmasked filters inside the DSM module, each are compared against the packet in parallel. The first byte of the filter is compared against the first byte of the packet record after the ERF header, refer to the *Endace Extensible Record Format* document for more information on the packet record format. It is important to note that for Ethernet packets there are two bytes of padding added immediately after the ERF header, these padding bytes are the first to be compared against the filter. Each filter outputs a Boolean `true or false` value that is provided to the lookup table for further classification.

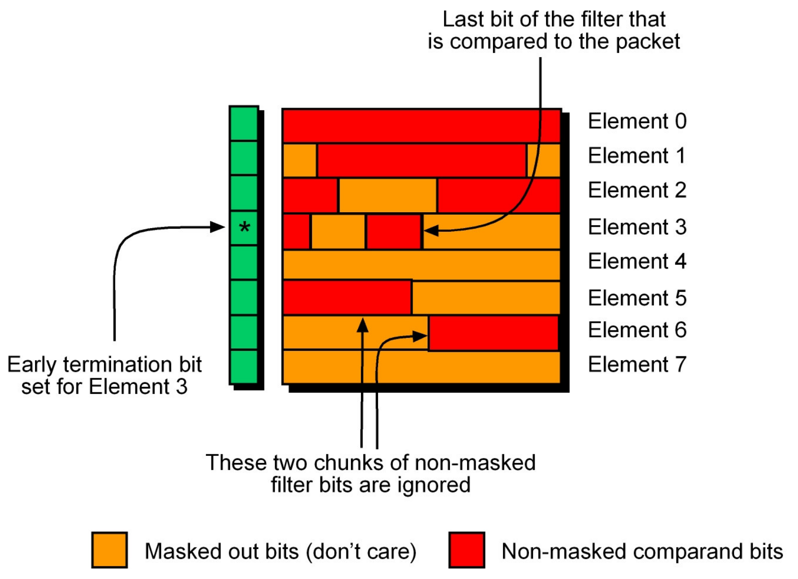
Filters also have an early termination option, this allows you to specify which 8-byte chunk (known as an element) of the filter contains the last byte to check. The early termination option is always specified on the last element in the filter (element 7). Packets that are smaller than the filter, as defined by the early termination option, always produce a `false` output regardless of the packet contents.

The diagram below shows a logical drawing of a filter. Each row represents 8 bytes of the filter (one element). In the diagram, the filter will be applied to the first 28 bytes (3 elements × 8 bytes + 4 non-masked bytes of element 3) of the packets rather than the full 64. Packets that are smaller than 28 bytes will produce a `false` output.


8 bytes
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5
Element 6
Element 7
*
Early termination bit set for Element 3
Last bit of the filter that is compared to the packet
Masked out bits (don't care)
Non-masked comparand bits

Packet Filters (cont.)

The diagram below shows that any non-masked bytes of the filter that occur in elements after the early termination option are effectively ignored regardless of the packet length.


Last bit of the filter that is compared to the packet
Element 0
Element 1
Element 2
Element 3
Element 4
Element 5
Element 6
Element 7
*
Early termination bit set for Element 3
These two chunks of non-masked filter bits are ignored
Masked out bits (don't care)
Non-masked comparand bits

Load Balancing Algorithms

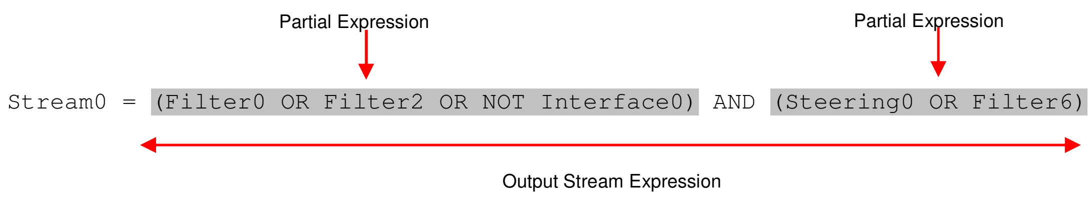
Two load balancing algorithms are applied to the packet, each resulting in a Boolean output value, both outputs are provided to the lookup table for further classification. The first algorithm is a CRC calculation applied to the expected location of an IPv4 packet's source and destination address within the packet record. The second algorithm calculates the parity, across the expected location of an IPv4 packet's source and destination addresses.

For a random collection of packet data, both algorithms give an approximately 50:50 split of `true and false` outputs. The load balancing algorithms are fully implemented in firmware and are not user configurable.

Lookup Table

The lookup table accepts the outputs from the filters, load balancing algorithm and the physical port number of the packet, to generate either a target stream number for the packet or a drop indication. The lookup table is fully user programmable, allowing for complex expressions to be constructed.

The DSM API provides a two stage implementation of the lookup table construction. The first stage involves creating one or more partial expressions, each parameter of the expression(or the inverse of the parameter) is logically OR'ed together to produce the partial expression. In the second stag, stream output expressions are constructed, containing one or more partial expressions, each partial expression (or the inverse of the partial expression) is AND'ed together.



```
Stream0 = (Filter0 OR Filter2 OR NOT Interface0) AND (Steering0 OR Filter6)
```

Load Balancing Algorithms (cont.)

Packet records can be routed to only one stream, if more than one output expression returns a Boolean `true` value for a set of input parameters, the stream with the highest priority (lowest stream number) will receive the packet record.

For example if the output stream expressions were the same for both stream 0 and stream 2, packet records that are accepted by the expression will only be routed to stream 0.

Output Record Format

Packets that are sent though the DSM are marked with a *color* value, this value encodes the outputs of the eight filters and two load balancing algorithms, as well as the target receive stream.

Refer to Appendix A for the format of ERF record header including the color field.

Counters

The DSM module maintains a minimum of thirteen counters, each counter is 32-bits and wraps back to zero on overflow.

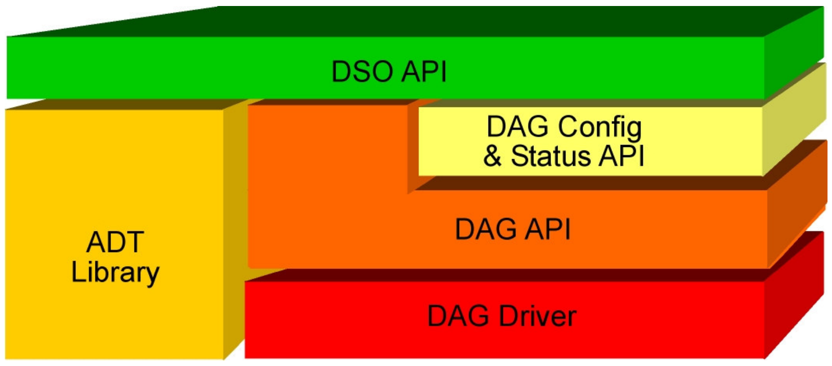
Type	Count	Description
Filter	8	Each filter has a counter indicating how many times the filter has output a true result.
Load Balancing	2	Both of the load balancing algorithms have a counter indicating how many true results have been generated
Drop	1	Counts the number of packets that have been dropped.
Stream	n*	Each receive stream has a counter indicating the number of packet records that have been routed to that stream.

Note:* The number of receive stream counters depends on the number of receive streams available on the card, currently this is 2.

DSM Configuration & Status API

Dependencies

Because the DSM Configuration and Status API (shortened to DSM API for the remainder of this document) reads and writes configuration information directly from the DAG card it requires the DAG driver to be running and it expects the DAG API library to be present. The diagram below shows the logical layering of the various libraries required by the DSM API.



Structure

The DSM API is divided into six logical sections, all the sections are contained within a single library file.

Name	Description
Card	Provides the functionality to query the card status and current configuration. Functions are also provided to update the configuration in raw mode, bypassing the constructs generated by the other sections
Configuration	Provides the general interface for constructing and querying a virtual DSM configuration. Within this section is a function to download the virtual configuration to the card.
Filters	Contains the functions to create and modify filters.
Partial Expressions	Contains functions to create partial expression. Multiple partial expressions are combined to construct a stream output expression.
Stream Output Expressions	Contains functions to create stream output expressions, these expressions are used to generate the classification lookup table.
Counters	Provides an interface to the DSM counters.

DSM Configuration & Status API (cont.)

Typical Usage

The following steps are typically taken to configure the card:

- **Step 1:** Configure the card to capture traffic using the DAG Configuration and Status API or the command line `dagconfig` program.
- **Step 2:** Query the card to determine the following:
  - if DSM is supported
  - the number of physical ports.
  - if the card is Ethernet or SONET (PoS).
  - the number of possible receive streams.
- **Step 4:** Prepare the card for DSM filtering using `dagdsm_prepare_card` which will configure the non-DSM modules on the card to be compatible with the DSM module.
- **Step 3:** Create an empty virtual configuration and populate it with the initial filter and expression settings.
- **Step 4:** Download the virtual configuration to the card. Optionally take the DSM module out of bypass to enable the filtering (by default bypass is enabled).
- **Step 5:** Enable packet reception.
- **Step 6:** If filters need to be changed while the packets are being received use the hot-swap option to ensure no packet misclassification. The lookup table can be changed at any time without packet misclassification.
- **Step 7:** When finished configuring the card the virtual configuration should be destroyed, this does not effect the current card configuration.

Multiple Threads

The DSM API library is not thread safe, users are required to wrap function calls, were appropriate, with their own thread safe mechanism (for example semaphores or mutexes).

Chapter 2: Card Configuration

Overview

This section of the API contains functionality to query the configuration of a DAG card. Some of the functionality is duplicated by the DAG Configuration and Status API, either API can be used.

Additionally there is functionality to directly interface with the DSM filters, this provides a raw interface to the underlying firmware on the card. It is not recommended to mix the raw filter functions with the virtual configuration functions contained in the virtual filter section.

`dagdsm_is_dsm_supported`

Purpose Returns whether the DSM functionality is supported by the card.

Declared In `dagdsm.h`

Prototype `int dagdsm_is_dsm_supported (int dagfd)`

Parameters → `dagfd`
DAG file descriptor provided by `dag_open`

Returns `0` if the DAG card can support DSM but the wrong firmware is loaded into it, `1` if DSM is supported and `-1` if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (function not supported by the card)

Comments This function should be called before any other in the DSM API library, it simply performs a sanity check to verify that DSM is supported by the DAG card in its current configuration.

Possible causes for this function to fail are:

- DSM firmware is not loaded into the FPGA, refer to the DAG card manual fro more information.
- The DAG card doesn't support DSM.

dagdsm_prepare_card

Purpose Prepares the DAG card for DSM operation.

Declared In `dagdsm.h`

Prototype `int dagdsm_prepare_card` (int dagfd)

Parameters → `dagfd`
DAG file descriptor provided by dag_open.

Returns 0 if the card was configured successfully otherwise -1 if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (function not supported by the card)

Comments This function configures the non-DSM modules on the DAG card to support DSM filtering. This function should be called to configure the DAG card prior to taking the DSM module out of bypass mode.

This function performs the following actions:

- Enables packet record steering, based on the DSM classification (by default packet records are routed to receive stream 0 only, regardless of the DSM classification)
- Enables packet record dropping per stream (by default if a single receive stream buffer is full, all receive streams will drop packet records)
- Updates the expected size of the packet record CRC field in the DSM to match the current card configuration.

dagdsm_bypass_dsm

Purpose Enables/disables the DSM bypass option.

Declared In `dagdsm.h`

Prototype `int dagdsm_bypass_dsm` (int dagfd, uint32_t bypass)

Parameters → `dagfd`
DAG file descriptor provided by `dag_open`.

→ `bypass`
A zero value disables bypass mode, a non-zero value enables the DSM bypass mode.

Returns Returns `0` if the bypass mode was enabled/disabled successfully, otherwise `-1` is returned. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (DSM function not supported by the card or incorrect firmware loaded)

Comments When the card is in DSM bypass mode, all packet records bypass the DSM module and are presented to the host directly. Bypass should be disabled for normal DSM functionality.

When the card initially powers up or is reset, bypass mode is enabled.

dagdsm_is_card_ethernet

Purpose Indicates if the card is configured for Ethernet.

Declared In `dagdsm.h`

Prototype `int dagdsm_is_card_ethernet` (int dagfd)

Parameters → `dagfd`
DAG file descriptor provided by `dag_open`.

Returns `0` if the card is not configured for Ethernet, `1` if configured for Ethernet and `-1` if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (DSM function not supported by the card)

Comments This function duplicates functionality provided by the DAG Configuration and Status API, however it is included in this library for completeness.

`dagdsm_is_card_sonet`

Purpose Indicates if the card is configured for sonet (PoS).

Declared In `dagdsm.h`

Prototype `int dagdsm_is_card_sonet` (int dagfd)

Parameters → `dagfd`
DAG file descriptor provided by `dag_open`.

Returns `0` if the card is not configured for SONET, `1` if configured for SONET and `-1` if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (DSM function not supported by the card)

Comments This function duplicates functionality provided by the DAG Configuration and Status API, however it is included in this library for completeness.

`dagdsm_get_port_count`

Purpose Returns the number of physical ports on the card.

Declared In `dagdsm.h`

Prototype `int dagdsm_get_port_count` (int dagfd)

Parameters → `dagfd`
DAG file descriptor provided by dag_open.

Returns A positive value indicates the number of physical ports (interfaces) on the card, -1 is returned if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)

Comments This function duplicates functionality provided by the DAG Config API, however it is included in this library for completeness.

`dagdsm_get_filter_stream_count`

Purpose Returns the number of receive streams available on the card.

Declared In `dagdsm.h`

Prototype `int dagdsm_get_filter_stream_count (int dagfd)`

Parameters → `dagfd`
DAG file descriptor provided by `dag_open`

Returns A positive value indicates the number of receive streams (filter streams) available, -1 is returned if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (DSM function not supported by the card or incorrect firmware loaded)

Comments This function duplicates functionality provided by the DAG API, however it is included in this library for completeness.

Refer to the *DAG Programming Guide* for more information on receive streams.

`dagdsm_is_filter_active`

Purpose Gets the activation status of a DSM filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_is_filter_active (int dagfd, uint32_t filter)`

Parameters → `dagfd`
DAG file descriptor provided by `dag_open`

→ `filter`
The number of the physical filter on the card, filter numbers start at 0 and go through to 7.

Returns `0` if the filter is not active, `1` if active and `-1` if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (DSM function not supported by the card or incorrect firmware loaded)
- `EINVAL` (invalid argument)

Comments This function returns the state of a particular filter on the card. There are eight filters in total.

It is important to note that the filter numbers supplied to this function directly match the filter numbers on the card, this is not necessarily the case when using the filter numbers in a virtual configuration, i.e. the virtual configuration filter numbers may not match the filter numbers on the card. The following example is erroneous, it assumes both the virtual configuration filter number and card filter number are the same.

```
const uint32_t filter = 1;

config_h =  dagdsm_create_configuration(dagfd);
filter_h = dagdsm_get_filter(config_h, filter);

dagdsm_filter_enable_filter(filter_h, 1);

dagdsm_load_configuration(config_h);

/* the following assertion is not correct and may fail */
assert ( dagdsm_is_filter_active(dagfd, filter) == 1 );
```

`dagdsm_activate_filter`

Purpose Activates a DSM filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_activate_filter (int dagfd, uint32_t filter, uint32_t activate)`

Parameters → `dagfd`
DAG file descriptor provided by `dag_open`.

→ `filter`
The number of the physical filter on the card, filter numbers start at 0 and go through to 7.

→ `activate`
A non-zero value activates the filter, a zero value deactivates the filter.

Returns `0` if the filter was activated/deactivated and `-1` if an error occurred. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (DSM function not supported by the card or incorrect firmware loaded)
- `EINVAL` (invalid argument)

Comments This function activates or deactivates a particular filter on the card. As with the `dagdsm_is_filter_active` function, this function works directly with the filters on the card, refer to the comments in section 3.1.8 for more information.

This function differs from `dagdsm_enable_filter` in that `dagdsm_enable_filter` enables a virtual filter in the configuration, it doesn't directly change hardware settings. The two functions should not be intermixed.

Deactivated filters always supply an output of false to the lookup table, regardless of the contents or length of the packet being compared.

`dagdsm_load_filter`

Purpose Loads a filter directly to the card

Declared In `dagdsm.h`

Prototype `int dagdsm_load_filter(int dagfd, uint32_t filter, uint32_t term_depth, uint8_t * value, uint8_t * mask, uint32_t size)`

Parameters

→ `dagfd`
DAG file descriptor provided by `dag_open`.

→ `filter`
The number of the filter.

→ `term_depth`
The element (8-byte chunk) that the filter can early terminate on.

→ `value`
Array of comparand bytes that are loaded into the filter. This value can be `NULL` if the `size` argument is also `0`.

→ `mask`
Array of mask bytes that are loaded into the filter. This value can be `NULL` if the `size` argument is also `0`.

→ `size`
The size of both the `value` and `mask` buffers in bytes.

Returns `0` if the filter was updated otherwise `-1` for an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (DSM function not supported by the card or incorrect firmware loaded)
- `EINVAL` (invalid argument)

Comments As with the `dagdsm_activate_filter` and `dagdsm_is_filter_active` functions, the filter argument refers to the hardware filter number not the virtual filter number specified in a virtual configuration.

The `term_depth` argument refers to the element that should have the early termination option set, refer to section 2.1.1 for more information. If all 64 bytes of the filter should be used, set `term_depth` to the `DSM_NO_EARLY` constant.

The zeroth byte of the `value` and `mask` array corresponds to the first byte of the packet record after the ERF header. If the `size` argument is less than the actual size of the filter in the firmware (64 bytes), the remaining bytes are padded with zeros in both the `value` and `mask` arrays.

> **Warning** For Ethernet packets there are two bytes of padding immediately after the ERF header, the first two bytes of the filter are compared with these padding bytes.

Internally this function deactivates the filter before loading the new values, afterwards if the filter was previously activated it will be reactivated prior to the function returning.

Calling this function with a `size` of `0`, will write zero values into all the mask and comparand bytes, effectively clearing the filter.

Chapter 3: DSM Virtual Configuration

Overview

Functions contained in this section of the API provides the ability to create and maintain a virtual DSM configuration. Changes made to the virtual configuration will not be reflected on the card until either the `dagdsm_load_configuration` or `dagdsm_do_swap_filter` functions are called.

Configuration Loading

Because the process of loading the virtual configuration into the card is not instantaneous, packets that are received during this process may be misclassified or dropped. If misclassified or dropped packets are unacceptable, consider putting the DSM module in bypass mode while the configuration is being updated.

The following table illustrates the situations where incorrect or dropped packets could occur, the actual probability of either situation is dependant on the previous value of the filters in the DSM and the new values being loaded into them.

Description	Possible Misclassified Packets	Possible Dropped Packets
First Configuration Load The first time a configuration is loaded into the card after boot-up (or after loading the Firmware).	Yes	Yes
Changed Filter Configuration If one or more filters in the virtual configuration are changed and then the new configuration is download to the card.	Yes	Yes
Changed Lookup Table (Expressions) Configuration If the lookup table has changed (by updating the partial and/or output stream expressions) and then the updated virtual connection is downloaded to the card.	No	No
Hot-Swapping a Filter See below for a description of 'hot-swapping	No	No

Virtual Filter Hot-Swapping

Hot-swapping is the process where one of the filters is replaced by a new filter without any packets being misclassified or dropped. It works by reserving one of the filters in the card as a swap filter, when asked to hot-swap, the API loads the new filter values into the reserved filter and enables it, the lookup table is then updated to reflect the position of the new filter, this occurs atomically on a packet boundary thereby ensuring no misclassification, finally the old filter is disabled.

This process works on the assumption that the virtual configuration has been loaded into the DAG card prior to using the hot-swap functions.

The following code snippet demonstrates the process of hot-swapping, it assumes a virtual configuration has already been created and loaded into the card.

```
DsmFilterH     swap_filter_h;
DsmFilterH     org_filter_h;
const uint32_t filter_num = 0;

...

/* get a handle to the swap filter */
swap_filter_h = dagdsm_get_swap_filter (config_h);

/* copy the current filter and update the ethertype */
org_filter_h = dagdsm_get_filter (config_h, filter_num);
dagdsm_filter_copy (swap_filter_h, org_filter_h);
dagdsm_filter_set_ethertype (swap_filter_h, 0x0800, 0xFFFF);

/* perform the hot-swap */
dagdsm_do_swap_filter (config_h, filter_num);
```

dagdsm_create_configuration

Purpose Creates a new blank virtual configuration.

Declared In `dagdsm.h`

Prototype `DsmConfigH dagdsm_create_configuration (int dagfd)`

Parameters → `dagfd`
DAG file descriptor provided by `dag_open`.

Returns A handle to the new configuration or `NULL` to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EBADF` (bad file descriptor)
- `ENOENT` (DSM function not supported by the card or incorrect firmware loaded)
- `ENOMEM` (memory allocation error)

Comments This function creates a new virtual configuration and returns a handle to it, more than one virtual configuration can be created per card.

The following is the default settings when a new configuration is created:

- All filters are disabled.

- The layer 2 protocol of the filter is set to match the DAG card settings.
- The layer 3 protocol is set to IPv4 and all filter fields are cleared.
- Raw mode for the filters is disabled.
- The filter early termination option is set to `DSM_NO_EARLYTERM`.
- All partial and output stream expressions are empty (this will drop all packets).

The virtual configuration should be destroyed by calling `dagdsm_destroy_configuration` once you have finished with it.

dagdsm_load_configuration

Purpose Loads a virtual connection into the DAG card.

Declared In `dagdsm.h`

Prototype `int dagdsm_load_configuration (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns `0` if the configuration was loaded successfully, `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)
- `ENOMEM` (memory allocation error)

Comments This function accepts a virtual configuration handle and loads the filters and lookup table from the virtual configuration into the card.

Warning: Because this process is not instantaneous packets received during the process may be misclassified or dropped, refer Configuration Loading earlier in this chapter for more information.

dagdsm_destroy_configuration

Purpose Destroys an existing virtual configuration.

Declared In `dagdsm.h`

Prototype `int dagdsm_destroy_configuration (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns `0` if the configuration was destroyed otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function destroys a virtual configuration, this has no effect on the actual configuration loaded into the card. Once this function returns the virtual configuration handle should be discarded, continuing to use it will result in unpredictable behaviour.

dagdsm_get_filter

Purpose Returns a handle to a virtual filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_get_filter (DsmConfigH config_h, uint32_t filter)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

→ `filter`
The number of the filter, this should be a value in the range of `0` to `6`.

Returns A handle to the virtual filter, `NULL` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function returns a handle to one of the seven virtual filters of a virtual configuration.

> **Warning:** The filter number is not necessarily related to the actual filter number on the card as used in the `dagdsm_is_filter_active`, `dagdsm_activate_filter` and `dagdsm_load_filter` functions.

dagdsm_is_etherne

Purpose Indicates whether the virtual configuration is for Ethernet or not.

Declared In `dagdsm.h`

Prototype `int dagdsm_is_ethernet (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns `0` if the virtual configuration is not configured for Ethernet, `1` if Ethernet is configured and -1 if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

`dagdsm_is_sonet`

Purpose Indicates whether the virtual configuration is for sonet or not.

Declared In `dagdsm.h`

Prototype `int dagdsm_is_sonet (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns `0` if the virtual configuration is not configured for SONET (PoS), `1` if Sonet (PoS) is configured and `-1` if an error occurred. Call `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

`dagdsm_get_swap_filter`

Purpose Returns a handle to the virtual swap filter.

Declared In `dagdsm.h`

Prototype `DsmFilterH dagdsm_get_swap_filter (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns A handle to the virtual swap filter, `NULL` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function doesn't clear the swap filter, therefore it is recommended that the swap filter is explicitly cleared (using `dagdsm_filter_clear`) prior to setting any values.

`dagdsm_do_swap_filter`

Purpose Swaps the specified filter on the card with the virtual swap filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_do_swap_filter (DsmConfigH config_h, uint32_t filter)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

→ `filter`
The number of the virtual filter to swap out, this argument should be in the range of `0` to `6`.

Returns `0` is returned to indicate success, `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)
- `ENOMEM` (memory allocation error)

Comments This function performs an atomic filter swap, the virtual swap filter should have been configured prior to performing the swap.

This function assumes, but doesn't check, that the virtual configuration has already been downloaded to the card, unpredictable DSM behaviour will occur it the current virtual configuration hasn't been downloaded to the card.

Warning: The `filter` argument supplied to this function is the virtual configuration filter number and is not necessarily related to the actual filter number on the card as used in the `dagdsm_is_filter_active`, `dagdsm_activate_filter` and `dagdsm_load_filter` functions.

`dagdsm_clear_expressions`

Purpose Destroys all partial expressions and resets the stream output expressions.

Declared In `dagdsm.h`

Prototype `int dagdsm_clear_expressions (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns `0` is returned to indicate success, `-1 is` returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function destroys all the partial expressions, any partial expression handles stored by the caller are no longer valid and should be discarded. Using partial expression handles after calling this function will result in unpredictable behaviour.

All output expressions for the streams are cleared. Cleared or empty stream output expressions, result in all packets being dropped for that stream.

dagdsm_create_partial_expr

Purpose Creates a new partial expression.

Declared In `dagdsm.h`

Prototype `DsmPartialExpH dagdsm_create_partial_expr (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns A handle to the new partial expression, `NULL` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)
- `ENOMEM` (memory allocation error)

Comments The handle returned by this function can be past to any of the functions in the partial expression section.

Partial expressions exist until `dagdsm_clear_expressions` is called. If lots of partial expressions are continuously being created, consider calling `dagdsm_clear_expressions` periodically to reduce memory usage.

dagdsm_get_partial_expr_count

Purpose Returns the number of partial expressions currently created.

Declared In `dagdsm.h`

Prototype `int dagdsm_get_partial_expr_count (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns A positive number is returned indicating the number of partial expressions, -1 is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments Returns the number of partial expressions created with `dagdsm_create_partial_expr` since the last call to `dagdsm_clear_expressions`.

dagdsm_get_partial_expr

Purpose Returns a handle to a partial expression.

Declared In `dagdsm.h`

Prototype `int dagdsm_get_partial_expr (DsmConfigH config_h, uint32_t index)`

Parameters → `config_h`
`Handle to a virtual configuration returned by`
dagdsm_create_configuration.
→ `index`
The index of the partial expression to retreive.

Returns A handle to the partial expression at the given index, NULL is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

- Possible error codes:
  `EINVAL` (invalid argument)

Comments The order of the partial expressions is not guaranteed, the first partial expression created might not be the expression at index `0`. To iterate over all partial expressions start at index `0` and iterate up to the number of partial expressions as returned by `dagdsm_get_partial_expr_count`.

dagdsm_get_output_expr_count

Purpose Returns the number of possible stream output expressions.

Declared In `dagdsm.h`

Prototype `int dagdsm_get_output_expr_count (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by
`dagdsm_create_configuration`.

Returns A positive number is returned indicating the number of possible output expressions, `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

- Possible error codes:
  `EINVAL` (invalid argument)

Comments There is one output expression per stream, therefore the returned value is also the number of possible receive streams. This function is equivalent to `dagdsm_get_filter_stream_count` for a given DAG card.

This function returns the number of output expressions, however the actual expression numbers are even, therefore the following code snippet is incorrect:

```
count = dagdsm_get_output_expr_count(config_h);
for (i=0; i<count; i++)
expr_h = dagdsm_get_output_expr(i);

instead do:

count = dagdsm_get_output_expr_count(config_h);
for (i=0; i<count; i++)
expr_h = dagdsm_get_output_expr(i * 2);
```

`dagdsm_get_output_expr`

Purpose Returns the handle to an output stream expression.

Declared In `dagdsm.h`

Prototype `DsmOutputExpH dagdsm_get_output_expression (DsmConfigH config_h, uint32_t rx_stream)`

Parameters → `config_h`

Handle to a virtual configuration returned by `dagdsm_create_configuration.`

→ `rx_stream`

The stream number to get the output expression for, this refers to the receive streams therefore all stream numbers should even.

Returns A handle to the output expression for the given stream, NULL is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments Stream output expressions are not created or destroyed they are valid while the virtual configuration is valid.

DAG card convention is that all receive streams have even numbers starting at `0` and all transmit streams have odd numbers starting at `1`. The DSM API maintains that convention and stream 0 refers to the first receive stream and stream `2` refers to the second and so forth. Therefore it is an error to supply an odd number for the `rx_stream` argument.

Chapter4: DSM Virtual Filter Configuration

Overview

The functions contained in this section are used to construct the virtual filters for loading into to the DAG card. Internally the library maintains the type of packets expected by each filter, for example Ethernet frames encapsulating IPv4/TCP packets. The type of the packet can be user defined down to the layer 2 protocol which is defined by the DAG card.

It is important to note that although specific fields of a packet can be specified in a filter, they may not correspond to the actual fields in the packet. This is because the API makes assumptions about where data fields are located inside the packet, these assumptions may be incorrect if things like IP options or MPLS shims are present, offsetting the location of the fields inside the packet.

Raw Mode

Each filter can be put in raw mode, this enables the caller to specify a raw array or comparands and masks to be loaded into the filter. The zeroth entry of the user supplied raw arrays correspond to the zeroth byte of the filter, this is compared against the first byte of a packet record after the ERF header.

Layer 2 Protocol Types

As mentioned above the layer 2 protocol is defined by the actual DAG card that is being configured, therefore it cannot be changed. The layer 2 protocol type can be queried by calling `dagdsm_is_ethernet` or `dagdsm_is_sonet`. The following table illustrates which functions are available for which layer 2 protocol modes.

Function	Layer 2 Protocol Type	
	SONET (PoS)	Ethernet
`dagdsm_filter_set_hdlc_header`	●	
`dagdsm_filter_set_ethertype`		●
`dagdsm_filter_set_mac_src_address`		●
`dagdsm_filter_set_mac_dst_address`		●
`dagdsm_filter_enable_vlan`		●
`dagdsm_filter_set_vlan_id`		●*

Note:* only valid if the VLAN option has been enabled by calling `dagdsm_filter_enable_vlan`

Layer 3 Protocol Types

The layer 3 protocol type of the packet should be defined prior to setting any of the data fields contained within the actual protocol. For example it is an error to call `dagdsm_set_ip_source` on a virtual filter without first calling `dagdsm_filter_set_layer3_type` with an IPv4 argument . By default when a virtual configuration is created all the filters are cleared and therefore no protocol types are defined (except the layer 2 protocol which is set by the DAG card).

The following table demonstrates which functions can be called for a particular configuration of layer 3 & 4 types.

Function	Layer 3 Protocol Type			
	IPv4			
dagdsm_filter_set_ip_source	●			
dagdsm_filter_set_ip_dest	●			
dagdsm_filter_ip_fragment	●			
dagdsm_filter_set_ip_hdr_length	●			
dagdsm_filter_set_ip_protocol	●			
	TCP	**UDP**	**ICMP**	
dagdsm_filter_set_src_port	●	●		
dagdsm_filter_set_dst_port	●	●		
dagdsm_filter_set_tcp_flags	●			
dagdsm_filter_set_icmp_code			●	
dagdsm_filter_set_icmp_type			●	

The following example illustrates how a typical IPv4 filter would be constructed, error checking has been removed for brevity.

Configuring a Filter for IPv4/TCP

```
const uint32_t filter = 0;
DsmConfigH     config_h;
DsmFilterH     filter_h;
struct in_addr addr;
struct in_addr mask;


/* create the new configuration */
...


/* get a handle to the filter */
filter_h = dagdsm_get_filter (config_h, filter);

/* ensure raw mode is disabled */
dagdsm_filter_set_raw_mode (filter_h, 0);

/* clear the filter contents */
dagdsm_filter_clear (filter_h);
```

```
/* enable the filter */
dagdsm_filter_enable (filter_h, 1);

/* configure for IPv4 */
dagdsm_set_layer3_type (filter_h, kIPv4);

/* set a source and destination filter */
inet_aton("192.168.0.0", &addr);
inet_aton("255.255.0.0", &mask);
dagdsm_filter_set_ip_source(filter_h, &addr, &mask);

/* set the layer 4 type to TCP */
dagdsm_filter_set_ip_protocol(filter_h, IPPROTO_TCP);

/* set the destination TCP port to filter on */
dagdsm_filter_set_dst_port(80, 0xFFFF);

...
```

dagdsm_filter_clear

Purpose Clears the contents of a virtual filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_clear (DsmFilterH filter_h)`

Parameters → `filter_h`
Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

Returns `0` if the filter was cleared, `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments When a filter is cleared the filter is modified in the following ways:

All filter fields are cleared (masks and values are set to zero)

- Layer 2 protocol type is unchanged.
- Layer 3 protocol type is set to kIPv4.
- Layer 4 protocol type is set to zero.
- Raw mode is disabled.
- The filter enabled/disabled state is unchanged.

A cleared filter is equivalent to the filter state when a virtual configuration is first created.

This function clears the layer 4 protocols/types, meaning that the functions to set the layer 4 data fields (TCP ports, ICMP types, etc) will return error codes.

dagdsm_filter_copy

Purpose Copies the contents of one virtual filter to another virtual filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_copy (DsmFilterH dst_filter_h, DsmFilterH src_filter_h)`

Parameters → `dst_filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`, this is the destination filter that is copied over.

→ `src_filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`, this is the source filter.

Returns `0` if the filter was copied otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function copies all the filter fields and protocol types from one filter to another.

It is possible to copy virtual filters from two different virtual configurations, however both filters must have the same layer 2 type (either Sonet/PoS or Ethernet).

The filter enabled/disabled state and raw mode state are also copied by this function.

dagdsm_filter_get_values

Purpose Copies the raw filter data to user supplied buffer.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_get_values (DsmFilterH filter_h, uint8_t * value, uint8_t * mask, uint32_t max_size)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

← `value`

Pointer to an array that will receive the comparand part of the filter.

← `mask`

Pointer to an array that will receive the mask part of the filter.

→ `max_size`

The maximum number of bytes that can be copied into both the `value` and `mask` arrays.

Returns A positive number indicating how many bytes were copied into both arrays, `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function copies all the filter values, up to the size of the filter or `max_size` depending on which one has the smallest value. The bytes copied into the two arrays are the same as what would be downloaded to the card if `dagdsm_load_configuration` was called.

The following example simply shows what you would expect to read back. It is assumed that the virtual configuration is for a Sonet card configuration. Error checking has been removed for brevity.

```
DsmConfigH config_h;
DsmFilterH filter_h;
uint8_t    values[4];
uint8_t    mask[4];




/* initialise the virtual configuration and get a filter
handle*/
...

assert (dagdsm_is_sonet(config_h) == 1);

/* clear and set the hdlc header filter */
dagdsm_filter_clear (filter_h);
dagdsm_filter_set_hdlc_hdr (filter_h, 0xFF030021, 0xFFFF00FF);

/* read the filter data */
dagdsm_filter_get_values (filter_h, values, mask, 4);

/* check the comparand values are correct */
assert(values[0] == 0xFF);
assert(values[1] == 0x03);
assert(values[2] == 0x00);
assert(values[3] == 0x21);

/* check the mask values are correct */
assert(mask[0] == 0xFF);
assert(mask[1] == 0xFF);
assert(mask[2] == 0x00);
assert(mask[3] == 0xFF);
```

dagdsm_filter_enable

Purpose Enables/disables a virtual filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_enable (DsmFilterH filter_h, uint32_t enable)`

Parameters → `filter_h`
Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `enable`
A non-zero value enables the filter, zero disables the filter.

Returns `0` if the virtual filter was enabled/disabled, `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function does NOT enable or disable the filter on the actual card, it simply sets the state of the filter in the virtual configuration. The state will not be reflected on the DAG card until `dagdsm_load_configuration` is called.

Disabled filters can still be used in partial expressions, however their output will always be false.

Disabling a filter still allows you to modify any of the filter fields using the virtual filter functions contained in this section.

dagdsm_filter_set_early_term_depth

Purpose Sets the first element that has the early termination option set.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_early_term_depth (DsmFilterH filter_h, uint32_t element)`

Parameters → `filter_h`
Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `element`
The element number to set the early termination option on, if no early termination is required set this parameter to `DSM_NO_EARLYTERM`.

Returns 0 if the early termination option was set, -1 is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments For a detailed description of the early termination option refer to Packet Filters earlier in this Guide.

If the `element` argument is larger than the last possible element in the filter, it is clipped to the last possible value.

dagdsm_filter_set_raw_mode

Purpose Sets or resets the raw mode of the filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_raw_mode (DsmFilterH filter_h, uint32_t enable)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `enable`

A non-zero value enables raw mode and zero disables raw mode

Returns `0` if raw mode was set, `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments When in raw mode the contents of the filter is set by the `dagdsm_filter_set_raw_filter` function, this allows for custom filters to be created.

dagdsm_filter_get_raw_mode

Purpose Gets the status of raw mode for a filter.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_get_raw_mode (DsmFilterH filter_h)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

Returns `1` if raw mode is set, 0 if not set and `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

dagdsm_filter_set_raw_filter

Purpose Sets the raw comparand and mask bytes of the virtual filter.

Declared In `dagdsm.h`

Prototype
```
int dagdsm_filter_set_raw_bytes (DsmFilterH filter_h, const
uint8_t * value, const uint8_t * mask, uint32_t size)
```

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `value`

Array that contains the raw bytes to load into the comparand part of the filter. The array must contain at least size number of bytes.

→ `mask`

Array that contains the raw bytes to load into the mask part of the filter. The array must contain at least size number of bytes.

→ `size`

The number of bytes in both the value and mask arrays that should be copied into the filter. Currently the maximum filter size is 64 bytes.

Returns `0` if the filter values were set otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function will fail if raw mode hasn't been enabled (by calling `dagdsm_filter_set_raw_mode`).

If the `size` argument is less than the actual size of the filter in firmware (64 bytes), the remaining bytes are padded with zeros in both the `value` and `mask` arrays. If the `size` is larger than the actual size it is trimmed.

The zeroth byte of the value and mask arrays, when loaded into the DAG card, is compared against the first byte after the ERF header. For Ethernet frames the DAG card inserts two bytes of padding after the ERF record, refer to Appendix A for more information.

`dagdsm_filter_enable_vlan`

Purpose Enables the VLAN option for filters on Ethernet cards.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_enable_vlan (DsmFilterH filter_h, uint32_t enable)`

Parameters → `filter_h`
Handle to a virtual filter returned by dagdsm_get_filter or `dagdsm_get_swap_filter`.

→ `enable`
A non-zero value enables the `VLAN` option, a zero value disables the `VLAN` option.

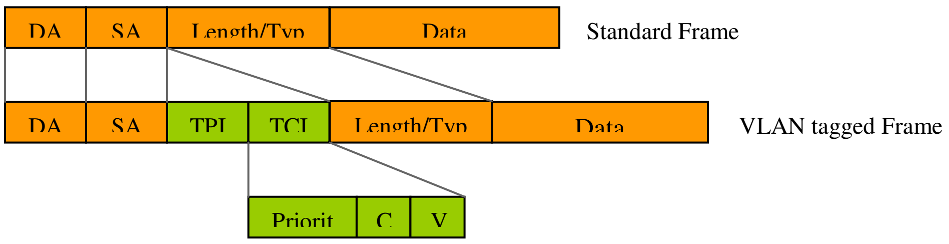
Returns 0 if the filter was updated otherwise -1 is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function will failed (with a `EINVAL` error code) if the DAG card is not configured for Ethernet.

This option is for IEEE 802.1Q tag-based VLAN only, the following diagram illustrates the `VLAN` format expected by the filter.



If `VLAN` is enabled the filter is adjusted to filter out Ethernet frames that have a length/type field of `0x8100` (this is the `TPID` field as per IEEE 802.1Q / 802.1P). All other data fields for the higher level protocols are automatically offset to the correct position.

If an Ethernet type/length filter field has been specified (by calling `dagdsm_filter_set_ethertype`) it is offset by four bytes to immediately after the `VLAN` header.

dagdsm_filter_set_vlan_id

Purpose Sets the VLAN ID to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_vlan_id (DsmFilterH filter_h, uint16_t id, uint16_t mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `id`

The 12-bit VLAN ID to filter on.

→ `mask`

The 12-bit mask of the id.

Returns `0` if the filter values were set otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

- Possible error codes:
  `EINVAL` (invalid argument)

Comments This function will fail with an `EINVAL` error code if the virtual configuration is not Ethernet with `VLAN` enabled.

The id and `mask` parameters should be in host byte order, internally the DSM API converts the value to network byte order.

dagdsm_filter_set_ethertype

Purpose Sets the ethertype of the Ethernet frame to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_ethertype (DsmFilterH filter_h, uint16_t ethertype, uint16_t mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `ethertype`

The Ethertype to filter on.

→ `mask`

The mask to use for the `Ethertype`.

Returns `0` if the filter values were set otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

- Possible error codes:
  `EINVAL` (invalid argument)

Comments This function will fail with an `EINVAL` error code if the virtual configuration is not configured for Ethernet. The `Ethertype` and mask parameters should be in host byte order, internally the DSM API converts the value to network byte order. This function doesn't have an effect on the type of the higher level protocol, for example calling this function with a E`thertype` argument of `0x0800`(the IPv4 protocol ethertype) is not equivalent to calling `dagdsm_filter_set_layer3_type` with the kIPv4 parameter.

`dagdsm_filter_set_mac_src_address`

Purpose Sets the source MAC address in the Ethernet header to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_mac_src_address (DsmFilterH filter_h, uint8_t src[6] uint8_t mask[6])`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `src`

The source MAC address to filter on.

→ `mask`

The mask to use for the MAC address.

Returns `0` if the filter values were set otherwise -1 is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function will fail with an EINVAL error code if the virtual configuration is not configured for Ethernet.

The MAC addresses bytes should be in network byte order (from most significant to least significant), for example to set a MAC address filter of `12:××:56:78:×A:BC` the following code should be used.

```
uint8_t addr[6] = { 0x12, 0x00, 0x56, 0x78, 0x0A, 0xBC };
uint8_t mask[6] = { 0xFF, 0x00, 0xFF, 0xFF, 0x0F, 0xFF };
dagdsm_filter_set_mac_src_address (filter_h, addr, mask);
```

dagdsm_filter_set_mac_dst_address

Purpose Sets the destination MAC address in the Ethernet header to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_mac_dst_address (DsmFilterH filter_h, uint8_t dst[6] uint8_t mask[6])`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter.`

→ `dst`

The destination MAC address to filter on.

→ `mask`

The mask to use for the MAC address.

Returns `0` if the filter values were set otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error t`o retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function will fail with an `EINVAL` error code if the virtual configuration is not configured for Ethernet.

Refer to the comments in previous section for more information on the usage of this function.

dagdsm_filter_set_hdlc_header

Purpose Sets the PoS HDLC/PPP 32-bit header to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_hdlc_header (DsmFilterH filter_h, uint32_t hdlc_hdr, uint32_t mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter.`

→ `hdlc_hdr`

The 32-bit HDLC/PPP PoS header to filter on.

→ `mask`

The mask to use for HDLC/PPP header.

Returns `0` if the filter values were set otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error t`o retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function will fail with an `EINVAL` error code if the virtual configuration is not configured for Sonet(PoS).

The `hdlc_hdr` argument should be in host byte order, internally the DSM API converts the parameter in network byte order.

dagdsm_filter_set_layer3_type

Purpose Sets the layer 3 protocol type to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_layer3_type (DsmFilterH filter_h, layer3_type_t type)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `type`

The layer 3 protocol type to filter on, currently the only valid value is kIPv4.

Returns `0` if the filter values were set otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This setting effects how the filter is constructed and how the higher level protocol data fields are used. Currently the only possible layer 3 protocol type is Internet Protocol version 4 (kIPv4).

Setting the layer 3 type doesn't change the actual filter bits, it just allows the IPv4 fields to be set.

dagdsm_filter_set_ip_protocol

Purpose Sets the IP protocol to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_ip_protocol (DsmFilterH filter_h, uint8_t type)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `protocol`

The 8-bit number that defines the IP protocol to filter on.

Returns `0` if the filter IP protocol were set otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function sets the protocol field of an IPv4 packet in the filter, there is no mask associated with this value so for a filter to hit on a packet, all eight bits of the protocol must match.

If the IP protocol number for either TCP(6), UDP(11) or ICMP(1) is specified in the protocol argument, additional protocol data fields can be configured, refer to table 3-3 for more information on possible function calls.

dagdsm_filter_set_ip_source

Purpose Sets the IPv4 source address to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_ip_source (DsmFilterH filter_h, struct in_addr *src, struct in_addr * mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `src`

Source address value to filter on.

→ `mask`

Source address mask.

Returns `0` if the filter was updated otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function updates the bits in the virtual filter to filter out packets with the correct source IP address.

In the following example, packets with an IP source address of `192.168.×.×` (where × refers to 'don't care' values) return a `true` output.

```
DsmFilterH     filter_h;
struct in_addr addr;
struct in_addr mask;


/* create the new configuration and get a handle to the filter*/
...

/* set a source and destination filter */
addr.s_addr = inet_addr("192.168.0.0");
mask.s_addr = inet_addr("255.255.0.0");
dagdsm_filter_set_ip_source(filter_h, &addr, &mask);
```

`dagdsm_filter_set_ip_dest`

Purpose Sets the IPv4 destination address to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_ip_dest (DsmFilterH filter_h, struct in_addr *dst, struct in_addr * mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `dst`

Destination address value to filter on.

→ `mask`

Destination address mask.

Returns `0` if the filter was updated otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments See the comments in previous section for more information.

`dagdsm_filter_set_ip_hdr_length`

Purpose Sets the IP header length to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_ip_hdr_length (DsmFilterH filter_h, uint8_t ihl)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `ihl`

The IP header length in 32-bit words to filter on, only the lower 4 bits of the value are used.

Returns `0` if the filter was updated, otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function updates the IP header length in the filter and adjusts any layer 4 filter fields to the correct offset. The minimum header length allowed is 5 words (40 bytes). There is no mask associated with this value so an exact value in the packet is required for a filter hit.

dagdsm_filter_ip_fragment

Purpose Updates the filter to reject IPv4 fragments.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_ip_fragment (DsmFilterH filter_h, uint32_t enable)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter.`

→ `enable`

A non-zero value will enable the IPv4 fragment rejection option in the filter. A zero value disables the option.

Returns `0` if the filter was updated otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function adds an entry in the filter to return false for IPv4 packets that have either the `More` bit set in the flags or the `Fragment Offset` field is not `0` in the packet header.

The following is the equation describing the filter output if IP fragment filtering is enabled:

```
filter output = (NOT more) AND (fragment offset = 0 )
```

dagdsm_filter_set_src_port

Purpose Sets the source port to filter on for TCP and UDP filters.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_src_port (DsmFilterH filter_h, uint16_t port, uint16_t mask)`

Parameters → `filter_h`

`Handle to a virtual filter returned by dagdsm_get_filter or dagdsm_get_swap_filter.`

→ `port`

The port number to filter on, this value should NOT be converted to network byte order, internally the library maintains the correct byte ordering of values.

→ `mask`

The mask value to use for the port, as with the port argument the mask should be in host byte order.

Returns `0` if the filter was updated otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments If the filter hasn't been configured for TCP or UDP (by calling `dagdsm_filter_set_ip_protocol` with a protocol argument of either 6 (TCP) or 17 (UDP)) this function will fail with a `EINVAL` error code.

The port and mask parameters should be in host byte order, internally the DSM API converts the value to network byte order.

In the following code snippet, UDP packets with a source port address of 80 (HTTP) will result in a true filter output, all other packets will result in a false output. Error checking has been omitted for brevity.

```
dagdsm_filter_set_ip_protocol (filter_h, 17);
dagdsm_filter_set_src_port (filter_h, 80, 0xFFFF);
```

dagdsm_filter_set_dst_port

Purpose Sets the destination port to filter on for TCP and UDP filters.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_dst_port (DsmFilterH filter_h, uint16_t port, uint16_t mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `port`

The port number to filter on, this value should not be converted to network byte order, internally the library maintains the correct byte ordering of values.

→ `mask`

The mask value to use for the port, as with the port argument the mask should be in host byte order.

Returns `0` if the filter was updated otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments If the filter hasn't been configured for TCP or UDP (by calling `dagdsm_filter_set_ip_protocol` with a protocol argument of either 6 (TCP) or 17 (UDP)) this function will fail with an `EINVAL` error code.

The port and mask parameters should be in host byte order, internally the DSM API converts the value to network byte order.

Refer to the comments in previous section for more information.

dagdsm_filter_set_tcp_flags

Purpose Sets the TCP flags to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_tcp_flags (DsmFilterH filter_h, uint8_t flags, uint8_t mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `flags`

The flags to filter on, only the lower 6 bits of the value are used.

→ `mask`

The mask to apply to the flags, only the lower 6 bits of the mask are used.

Returns `0` if the filter was updated, otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments If the filter hasn't been configured for TCP (by calling `dagdsm_filter_set_ip_protocol` with a protocol parameter of 6 (TCP)) this function will fail with an `EINVAL` error code.

In the following example all IPv4/TCP packets that have the RST (reset connection) and PSH (push function) TCP flags set, will result in a true filter output, all other packets will result in a false filter output. Error checking has been omitted for brevity.

```
dagdsm_filter_set_ip_protocol (filter_h, 6);
dagdsm_filter_set_tcp_flags (filter_h, 0x0C, 0x0C);
```

dagdsm_filter_set_icmp_code

Purpose Sets the ICMP code to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_icmp_code (DsmFilterH filter_h, uint8_t code, uint8_t mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `code`

The ICMP code to filter on.

→ `mask`

The mask to apply to the code.

Returns `0` if the filter was updated, otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments If the filter hasn't been configured for ICMP (by calling `dagdsm_filter_set_ip_protocol`, with a protocol parameter of 1 (ICMP)) this function will fail with a `EINVAL` error code.

In the following example all IPv4/ICMP packets that have a code of `0x12`, will result in a true filter output, all other packets will result in a false filter output. Error checking has been omitted for brevity.

```
dagdsm_filter_set_ip_protocol (filter_h, 1);
dagdsm_filter_set_tcp_flags (filter_h, 0x12, 0xFF);
```

dagdsm_filter_set_icmp_type

Purpose Sets the ICMP type to filter on.

Declared In `dagdsm.h`

Prototype `int dagdsm_filter_set_icmp_type (DsmFilterH filter_h, uint8_t type, uint8_t mask)`

Parameters → `filter_h`

Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

→ `type`

The ICMP type to filter on.

→ `mask`

The mask to apply to the type.

Returns `0` if the filter was updated, otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments If the filter hasn't been configured for ICMP (by calling `dagdsm_filter_set_ip_protocol` with a protocol parameter of 1 (ICMP)) this function will fail with an `EINVAL` error code.

In the following example all IPv4/ICMP packets that are an echo request (type `8`), will result in a `true` filter output, all other packets will produce a false filter output. Error checking has been omitted for brevity.

```
dagdsm_filter_set_ip_protocol (filter_h, 1);
dagdsm_filter_set_tcp_flags (filter_h, 8, 0xFF);
```

Chapter 5: Partial Expressions

Overview

The functions contained in this section are used to construct logical partial expressions, the expressions are used to generate the lookup table in the DSM module. Partial expressions are constructed from one or more; filter outputs, load balancing algorithm outputs or physical port numbers, each of the values (or the inverse of the values) are OR'ed together to create the partial expression. The output of a partial expression is a boolean value. If a partial expression contains no parameters it will always return false.

It is not possible to create a partial expression that contains a parameter and the inverse of itself. For example if erroneously trying to create a partial expression with parameters of; *Filter0* and *not Filter0*, the following code might be used:

```
partial_h = dagdsm_create_partial_expr (config_h);
dagdsm_expr_set_filter (partial_h, 0, 0);
dagdsm_expr_set_filter (partial_h, 0, 1);
```

However the actual partial expression that is created will just have a single parameter of *not Filter0*, because that was the last value set for filter 0.

Below are example partial expressions and the source code required to construct them. Both examples assume a virtual configuration has been created. Error checking has been removed for brevity.

Partial Expression Example 1

```
output= Filter0 OR NOT Filter1
partial_h = dagdsm_create_partial_expr (config_h);
dagdsm_expr_set_filter (partial_h, 0, 0);
dagdsm_expr_set_filter (partial_h, 1, 1);
```

Partial Expression Example 2

```
output = Physical Port0 OR NOT Filter6 OR NOT Steering0 OR
Filter2
partial_h = dagdsm_create_partial_expr (config_h);
dagdsm_expr_set_interface (partial_h, 0, 0);
dagdsm_expr_set_filter (partial_h, 6, 1);
dagdsm_expr_set_hlb (partial_h, 0, 1);
dagdsm_expr_set_filter (partial_h, 2, 0);
```

dagdsm_expr_set_filter

Purpose Sets the filter parameter in a partial expression.

Declared In `dagdsm.h`

Prototype `int dagdsm_expr_set_filter (DsmPartialExpH expr_h, uint32_t filter, uint32_t invert)`

Parameters → `expr_h`

Handle to a partial expression returned by either `dagdsm_create_partial_expr` or `dagdsm_get_partial_expr`.

→ `filter`

The virtual filter number to set in the expression.

→ `invert`

A non-zero value will invert the filter parameter in the expression, zero will not invert the parameter.

Returns `0` if the partial expression was updated otherwise `-1` is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

dagdsm_expr_set_interface

Purpose Sets the physical interface(port) parameter in a partial expression.

Declared In `dagdsm.h`

Prototype `int dagdsm_expr_set_interface (DsmPartialExpH expr_h, uint32_t iface, uint32_t invert)`

Parameters → `expr_h`

Handle to a partial expression returned by either `dagdsm_create_partial_expr` or `dagdsm_get_partial_expr`.

→ `iface`

The physical interface number to set in the partial expression.

→ `invert`

A non-zero value will invert the interface parameter in the expression, a zero value will not invert the parameter.

Returns 0 if the partial expression was updated otherwise -1 is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function sets the physical port or interface number to add to the partial expression. The physical interface is recorded by the DAG card when the packet first arrives at the card, interfaces are number 0 through to n-1 where n is the number of physical interfaces.

This function will fail (with an error code of `EINVAL`) if the `iface` argument is greater than the number of physical ports on the card.

`dagdsm_expr_set_hlb`

Purpose Sets the load balancing (steering) parameter in a partial expression.

Declared In `dagdsm.h`

Prototype `int dagdsm_expr_set_hlb (DsmPartialExpH expr_h, uint32_t hlb, uint32_t invert)`

Parameters → `expr_h`

Handle to a partial expression returned by either dagdsm_create_partial_expr or dagdsm_get_partial_expr.

→ `hlb`

The number of the load balancing algorithm to use as a parameter in the partial expression. The following constants are defined and can be used for this argument:

`kCRCLoadBalAlgorithm`

`kParityLoadBalAlgorithm`.

→ `invert`

A non-zero value will invert the load balancing parameter in the expression, a zero value will not invert the parameter.

Returns 0 if the partial expression was updated otherwise -1 is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function sets the load balancing (steering) algorithms that will be used in a partial expression. Each algorithm will produce on average a 50:50 `true/false` output, given random packet data. Therefore these algorithms can be used in a partial expression to split packet records between two different receive streams.

`dagdsm_compute_partial_expression`

Purpose Calculates the output of a partial expression given a complete set of input parameters.

Declared In `dagdsm.h`

Prototype

```
int dagdsm_compute_partial_expression (DsmPartialExpH expr_h,
uint32_t filters, uint8_t iface, uint32_t hlb0, uint32_t hlb1)
```

Parameters

→ `expr_h`

Handle to a partial expression returned by either `dagdsm_create_partial_expr` or `dagdsm_get_partial_expr`.

→ `filters`

Bitmasked value that should contain the filters expected to hit, see the comments below for more information.

→ `iface`

The interface number to check against, only the lower 2 bits of this value are used.

→ `hlb0`

A non-zero value to indicate the output of the CRC load balancing algorithm is true, a zero value indicates the output is false.

→ `hlb1`

A non-zero value to indicate the output of the parity load balancing algorithm is true, a zero value indicates the output is false.

Returns 0 if partial expression evaluates to a false output, 1 if the partial expression evaluates to a true output and -1 to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function computes the output of a given partial expression, based on the supplied parameters, this may be useful for checking partial expression logical, before downloading the configuration to the card.

The filter parameter is a bit-masked value indicating the filter outputs to compare with the partial expression. The zeroth bit corresponds to filter 0, bit 1 corresponds to filter 1 and so on, a maximum of 7 bits can be set. For example if `0x00000045` was supplied as the filters argument, it would indicate that filters 0, 2 & 6 have produced a true output and all the other filters have produced a false output.

Partial expressions that have no parameters set always return a false output, regardless of the filters, interface and load balancing input parameters.

Chapter 6: Output Expressions

Overview

Output expressions are constructed of one or more partial expressions (or the inverse of the partial expression) AND'ed together. A single output expression corresponds to a single receive stream, output expressions cannot be created or destroyed. To clear the contents of an output expression call `dagdsm_clear_expressions`.

As with the partial expressions, if an output expression has no partial expression parameters it will always output a false value.

It is possible to create an output expression that will never output a `true` value regardless of input parameters, the following example illustrates the situation.

```
partial expression 0 = filter 0
partial expression 1 = filter 0
output expression  = partial expression 0 AND NOT partial
expression 1
```

The DSM API doesn't check for this situations, it is the user responsibility to correctly construct the output expressions.

Below are some example stream output expressions and the source code used to generate them. In each case, error checking has been removed for brevity.

```
output = expression0 AND expression1
DsmOutputExpH  output_h;
DsmPartialExpH partial0_h;
DsmPartialExpH partial1_h;

output_h = dagdsm_get_output_expression (config_h, 0);
dagdsm_expr_add_partial_expr (output_h, partial0_h, 0);
dagdsm_expr_add_partial_expr (output_h, partial1_h, 0);
```

Stream Output Expression Example 2

output = expression0 AND NOT expression1 AND NOT expression2

DsmOutputExpH output_h;

DsmPartialExpH partial0_h;

DsmPartialExpH partial1_h;

DmsPartialExpH partial2_h;

output_h = dagdsm_get_output_expression (config_h, 0);

dagdsm_expr_add_partial_expr (output_h, partial0_h, 0);

dagdsm_expr_add_partial_expr (output_h, partial1_h, 1);

dagdsm_expr_add_partial_expr (output_h, partial2_h, 1);

`dagdsm_expr_add_partial_expr`

Purpose Adds a partial expression to the output expression.

Declared In `dagdsm.h`

Prototype `int dagdsm_expr_set_hlb (DsmOutputExpH expr_h, DsmPartialExpH partial_h, uint32_t invert)`

Parameters → `output_h`

Handle to an output expression returned by `dagdsm_get_output_expression`.

→ `partial_h`

Handle to a partial expression returned by either `dagdsm_create_partial_expr` or `dagdsm_get_partial_expr`.

→ `invert`

A non-zero value will invert the output of the partial expression, a zero value will not invert the partial expression output.

Returns 0 if the output expression was updated otherwise -1 is returned to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

- Possible error codes:
  `EINVAL` (invalid argument)

Comments This function appends a partial expression (or the inverse of a partial expression) onto the end of the output expression. The partial expression must belong to the same virtual configuration that holds the output expression, unpredictable behaviour will result if expressions are used across different virtual configurations.

The invert argument allows for the output of the partial expression to be inverted prior to being AND'ed with the rest of the output expression.

`dagdsm_compute_output_expr_value`

Purpose Computes the output value of an output expression, given a fixed set of input parameters.

Declared In `dagdsm.h`

Prototype `int dagdsm_compute_output_expr_value (DsmOutputExpH expr_h, Duint32_t filters, uint8_t iface, uint32_t hlb0, uint32_t hlb1)`

Parameters → `expr_h`

Handle to an output expression returned by `dagdsm_get_output_expression`.

→ `filters`

Bit-masked value that should contain the filter outputs, one bit per filter, see the comments below for more information.

→ `iface`

The interface number to check against, only the lower 2 bits of this value are used.

→ `hlb0`

A non-zero value to indicate the output of the CRC load balancing algorithm is true, a zero value indicates the output is false.

→ `hlb1`

A non-zero value to indicate the output of the parity load balancing algorithm is true, a zero value indicates the output is false.

Returns 0 if the stream output expression evaluates to a false output, 1 if the stream output expression evaluates to a true output and -1 to indicate an error. Use dagdsm_get_last_error to retrieve the error code.

Possible error codes:

- EINVAL (invalid argument)

Comments This function computes the output of a given stream output expression, based on the supplied parameters, this may be useful for checking stream output expression logic, before downloading the configuration to the card.

The filter parameter is a bit-masked value indicating the filter outputs to compare with the partial expression. The zeroth bit corresponds to filter 0, bit 1 corresponds to filter 1 and so on, a maximum of 7 bits can be set. For example if 0x00000045 was supplied as the filters argument, it would indicate that filters 0, 2 & 6 have a true output and all the other filters have a false output.

Stream output expressions that have no partial expression parameters, always return a false output regardless of the filters, interface and load balancing input parameters.

Warning: The output of this function is not necessarily the value that will be programmed into the lookup table on the card. This is because output stream expressions are priority ordered, meaning that if the same set of input parameters gives hits on more than one stream expression, the stream expression with the highest priority (lowest stream number) is programmed into the table.

Chapter 7: Counters

Overview

The functions contained in this section are used to latch and clear all the counters and read back the latched values. The counters should be latched and cleared prior to reading the values.

`dagdsm_latch_and_clear_counters`

Purpose Latches and clears all the counters inside the card.

Declared In `dagdsm.h`

Prototype `int dagdsm_latch_and_clear_counters (DsmConfigH config_h)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

Returns `0` if the counters were latched and cleared otherwise `-1` to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Comments This function latches the counters inside the DAG card and copies the values into registers accessible from the API. When a counter is read the latched values are returned.

`dagdsm_read_filter_counter`

Purpose Reads the contents of a latched filter counter.

Declared In `dagdsm.h`

Prototype `int dagdsm_read_filter_counter (DsmConfigH config_h, DsmFilterH filter_h, uint32_t *value_p)`

Parameters → `config_h`
Handle to a virtual configuration returned by `dagdsm_create_configuration`.

→ `filter_h`
Handle to a virtual filter returned by `dagdsm_get_filter` or `dagdsm_get_swap_filter`.

← `value_p`
Pointer to a 32-bit variable that receives the latched filter count.

Returns `0` if the latched counter value was read otherwise `-1` to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

`dagdsm_read_hlb_counter`

Purpose Reads the contents of a latched load balancing algorithm counter.

Declared In `dagdsm.h`

Prototype `int dagdsm_read_hlb_counter (DsmConfigH config_h, uint32_t *hlb0_p, uint32_t *hlb1_p)`

Parameters → `config_h`

Handle to a virtual configuration returned by `dagdsm_create_configuration`.

← `hlb0_p`

Pointer to a 32-bit variable that receives the latched CRC load balancing algorithm count. This is an optional output, pass `NULL` if this counter value is not required.

← `hlb1_p`

Pointer to a 32-bit variable that receives the latched parity load balancing algorithm count. This is an optional output, pass `NULL` if this counter value is not required.

Returns `0` if the latched counter value was read otherwise `-1` to indicate an error. Use `dagdsm_get_last_error t`o retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

`dagdsm_read_drop_counter`

Purpose Reads the contents of the latched drop counter.

Declared In `dagdsm.h`

Prototype `int dagdsm_read_drop_counter (DsmConfigH config_h, uint32_t *value_p)`

Parameters → `config_h`

Handle to a virtual configuration returned by `dagdsm_create_configuration`.

← `value_p`

Pointer to a 32-bit variable that receives the latched drop count.

Returns `0` if the latched counter value was read otherwise `-1` to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

`dagdsm_read_stream_counter`

Purpose Reads the contents of the latched packet counter for a particular receive stream.

Declared In `dagdsm.h`

Prototype `int dagdsm_read_stream_counter (DsmConfigH config_h, uint32_t stream, uint32_t *value_p)`

Parameters → `config_h`

Handle to a virtual configuration returned by `dagdsm_create_configuration.`

→ `stream`

The receive stream to read the packet count from.

← `value_p`

Pointer to a 32-bit variable that receives the latched stream packet count.

Returns `0` if the latched counter value was read otherwise `-1` to indicate an error. Use `dagdsm_get_last_error` to retrieve the error code.

Possible error codes:

- `EINVAL` (invalid argument)

Chapter 8: Miscellaneous Functions

Overview

This section contains functions that don't fit into one of the other logic sections.

`dagdsm_get_last_error`

Purpose	Reads the contents of the latched drop counter.
Declared In	`dagdsm.h`
Prototype	`int dagdsm_get_last_error (void)`
Parameters	none
Returns	The last error code generated by one of the `dagdsm_` functions, refer to the function in question for possible error codes.
Comments	When any of the `dagdsm_` functions are called, they internally reset the last error value to 0, therefore the last error code will not persist across multiple DSM function calls.

Chapter 9: Troubleshooting

Reporting Problems

If you have problems with a DAG card or Endace supplied software which you are unable to resolve, please contact Endace Customer Support at support@endace.com.

Supplying as much information as possible enables Endace Customer Support to be more effective in their response to you. The exact information available to you for troubleshooting and analysis may be limited by nature of the problem. However the following items will assist a quick resolution:

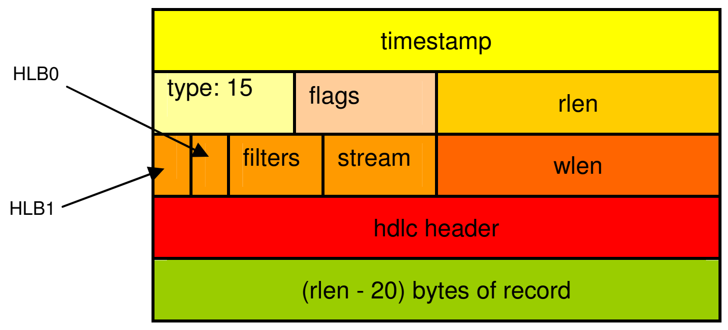
- DAG card[s] model and serial number.
- Host PC type and configuration.
- Host PC operating system version
- DAG software version package in use
- Any compiler errors or warnings when building DAG driver or tools
- For Linux and FreeBSD, messages generated when DAG device driver is loaded. These can be collected from command `dmesg`, or from log file `/var/log/syslog`.
- Output of daginf
- Firmware versions from `dagrom -x`.
- Physical layer status reported by: `dagconfig`
- Network link statistics reported by: `dagconfig -si`
- Extended link statistics reported by: `dagconfig -ei`
- Network link configuration from the router where available.
- Contents of any scripts in use.
- Complete output of session where error occurred including any error messages from DAG tools. The `typescript` Unix utility may be useful for recording this information.
- A small section of captured packet trace illustrating the problem.

Appendix A: ERF Record Format

Introduction

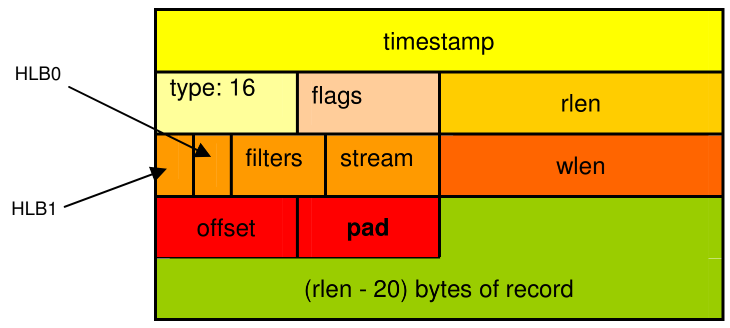
The following ERF record formats are generated by the DSM module, refer to the *EDM11-01 Endace Extensible Record Format* document for more detailed information.

Type 15 DSM Colored PoS HDLC Record



Data Format	Size	Description
timestamp	64 bits	The time of arrival of the cell, an ERF 64-bit timestamp, described in more detail in the *EDM11-01* document.
type	8 bits	This field contains an enumeration of the frame subtype. For DSM colored PoS records this value should be 15.
flags	8 bits	This byte is divided into 2 parts, the interface identifier, and a set of 1-bit flags.
rlen	16 bits	Record length. Total length of the record transferred over PCI bus to storage.
hlb1 & hlb0	1 bit each	Contains the output of each of the load balancing algorithms. `hlb0` = CRC algorithm output. `hlb1` = Parity algorithm output.
filters	8 bits	Each bit represents an output from one of the filters. `1` indicates a `true` output, `0` indicates a `false` output. Filter `0` is bit `0` of this field.
stream	6 bits	The target receive stream for the packet record.
wlen	16 bits	Wire length. The length of the packet that was received from the line.
hdlc header	32 bits	The 32-bit PoS frame HDLC header that was received from the line.

Type 16 DSM Ethernet Colored Record



Data Format	Size	Description
timestamp	64 bits	The time of arrival of the cell, an ERF 64-bit timestamp, described in more detail in the *EDM11-01* document.
Type	8 bits	This field contains an enumeration of the frame subtype. For DSM colored Etherent records this value should be 16.
flags	8 bits	This byte is divided into 2 parts, the interface identifier, and a set of 1-bit flags.
rlen	16 bits	Record length. Total length of the record transferred over PCI bus to storage.
hlb1 & hlb0	1 bit each	Contains the output of each of the load balancing algorithms. `hlb0` = CRC algorithm output. `hlb1` = Parity algorithm output.
filters	8 bits	Each bit represents an output from one of the filters. `1` indicates a `true` output, `0` indicates a `false` output. Filter `0` is bit `0` of this field.
stream	6 bits	The target receive stream for the packet record.
wlen	16 bits	Wire length. The length of the packet that was received on the line.
offset	8 bits	This field is currently not implemented, contents can be disregarded.
pad	8 bits	Padding byte so the Ethernet frame is aligned on a 16-bit boundary.

Version History

The version history for this programming guide is shown below.

Version	Date	Reason
1	February 2006	First release
2	August 2006	Support for DAG 8.2X and DAG 5.2X Format and layout changes .