endace

accelerated

HDLC Filtering Guide

www.endace.com

Table of Content

1. Purpose 4
1.1 Revision History 4
2. Introduction 5
2.1 Overview of the DAG 3.7T HDLC filtering 5
2.2 Notes on DAG3.7T specific functions 5
3. Functional Overview of the DAG 3.7T HDLC filtering 6
3.1 Software ID 6
3.2 Version ID 6
3.3 Temperature Sensor 6
3.4 Library and XScale Initialization 6
3.5 Default behavior of the HDLC Embedded Software 6
3.6 Statistics 7
3.7 Filtering 7
General Filtering 7
Extended Filtering 7
Extended Filtering Examples 7
4. DAG EMA Library Function Definitions 10
4.1 dagema_reset_processor 10
Description 10
Return Codes 10
4.2 dagema_open_conn 11
Description 11
Return Codes 11
4.3 dagema_close_conn 12
Description 12
Return Codes 12
4.4 dagema_get_last_error 13
Description 13
Return Codes 13
5. DAG 3.7T Specific Library Function Definitions 14
5.1 d37t_write_software_id 14
Description 14
Return Codes 14
5.2 d37t_read_software_id 15
Description 15
Return Codes 15
5.3 d37t_read_version 16
Description 16
Return Codes 16
5.4 d37t_read_temperature 17
Description 17
Return Codes 17
6. HDLC filtering Msg Function Definitions 18
6.1 hdlc_msg_set_burst_size 18
Description 18
Return Codes 18
6.2 aal_msg_flush_burst_buffer 19
Description 19
Return Codes 19
6.3 hdlc_msg_set_filter 20
Description 20
Return Codes 21

6.4 hdlc_msg_set_subfilter 22
Description 22
Return Codes 23
6.5 hdlc_msg_set_filter_action 24
Description 24
Return Codes 24
6.6 hdlc_msg_reset_filter 25
Description 25
Return Codes 25
6.7 hdlc_msg_reset_all_filters 26
Description 26
Return Codes 26
7. DAG Sar Function Definitions 27
Note: All the functions are used the same way as the AAL reassembly. And shared on the host side the same api. 27
7.1 dagsar_get_stats 27
Description 27
Return Codes 27
DAG3.7T Specific Return Codes 27
7.2 dagsar_get_interface_stats 28
Description 28
Return Codes 28
DAG3.7T Specific Return Codes 28
7.3 dagsar_reset_stats 29
Description 29
Return Codes 29
DAG3.7T Specific Return Codes 29
7.4 dagsar_reset_stats_all 30
Description 30
Return Codes 30
DAG3.7T Specific Return Codes 30
7.5 dagsar_set_filter_bitmask 31
Description 31
Examples 32
Return Codes 32
DAG3.7T Specific Return Codes 32
7.6 dagsar_reset_filter_bitmask 33
Description 33
Return Codes 33
DAG3.7T Specific Return Codes 33
8. Data Structures, Attributes, Defines and Enums 34
8.1 stats_t 34
8.2 filter_action_t 34
8.3 filter_operations_t 34
8.4 dag_filter_level_t 35
8.5 list_operations_t 35
9. Data Formats 36
9.1 Generic Data Format 36
9.2 Multichannel HDLC ERF Record 37
10. Examples 38
10.1 Layer 2 filtering FISU's, LSSU's and MSU's filtering example 38
10.2 Set up and close down of card and communications 45
10.3 Get Information about card 47
10.4 AAL Filtering if needed HDLC just replace the aal_ with hdlc_ in the filter setup 48
10.5 Statistics 50

1. Purpose

This document describes the functionality provided by the HDLC filtering embedded software available for the DAG3.7T card. Please be aware that this document is subject to change as additional functionality becomes available. This document may also be altered significantly to describe functionality available on future Endace DAG Cards.

1.1 Revision History

Rev.	**Date of Change**	**Description of Change**	**Revision Originator**
0.1	20/03/06	Initial version based on the AAL/SAR API documentation from Cassandra	Vladimir
0.2	17/07/06	Changes in HDLC Filtering interface due work with AAL(ATM) filtering in mixed mode image	Vladimir
0.2	31/07/06	Fixed examples	Vladimir

2. Introduction

2.1 Overview of the DAG 3.7T HDLC filtering

The DAG3.7T HDLC filtering software is designed to allow filtering on HDLC packets and Layer 2 filtering on the 3.7T card without involving the host in processing. The filter will receive HDLC traffic from the lines, this traffic is then either sent to the host unchanged or dropped, depending on the configuration used. Depending on the firmware and setup non HDLC (frame relay packets) or for similar protocols is used as well.

All the filtering and DAG3.7T specific functions are using the same SAR API functionality from the host side . With exception of the specific AAL functions. The information in this document will repeat parts of the SAR API documentation, but that is for easy use and independent use of the document.

2.2 Notes on DAG3.7T specific functions

Function definitions are described in later chapters of this document. Functions which begin with the prefix *d37t* are available from the DAG3.7T specific library. These functions will be similar, where possible, across different embedded software for the 3.7T, for example, functions for receiving the Software ID will be similar for AAL reassembly, and for IMA, and for HDLC

Function definitions beginning with the prefix *ema* are available from the embedded messaging library. These are similar, where possible, over different Endace DAG cards, and different forms of embedded software.

Functions beginning with the prefix *dagsar* are available from the SAR library. These functions will be similar across different DAG cards.

Functions beginning with the prefix hdlc*_msg* are also available from the dagsar library, but are specific to the Endace DAG 3.7T card. this functions are used and for HDLC due the similar functionalities.

However, while consistency has been a central part of the design philosophy, it is important to consult the relevant documentation to identify any differences.

3. Functional Overview of the DAG 3.7T HDLC filtering

3.1 Software ID

The DAG3.7T Software ID feature provides a mechanism to store and retrieve persistent customer-specific identification data on the board. The data is physically stored in EEPROM or Flash ROM medium, and as such, persists after the board is powered down.

Functions are provided to read and write the data. The **d37t_write_software_id()** function provides some protection, by requiring a 32-bit key to be used to enable write-access to the ID.

The DAG3.7T provides 128 bytes of storage for the Software ID. These can be used to implement a custom ID on the board for tracking purposes, bit-fields to enable or disable software features in the application, or any other custom use for the application developer.

3.2 Version ID

The DAG 3.7T Version ID feature allows the type of embedded software and the version of the embedded software to be queried. The type can be AAL, IMA, HDLC and version numbers can be used to determine which features are available.

3.3 Temperature Sensor

The DAG3.7T board is equipped with an LM63 temperature sensor attached to the XScale processor. The **d37t_read_temperature()** function allows the application to sample the current temperature reading from the board.

3.4 Library and XScale Initialization

The XScale processor is started and data is routed through the XScale to the host. First, to route the data correctly, use the **dag_set_mux()** function in the 3.7t specific library. The XScale can then be started by calling the **dagema_reset_processor()** function. **dagema_open_conn()** will initialize communications with the board, and set up a system to maintain it. **dagema_close_conn()** is used to end communications, and free the associated resources.

Note: this presumes that you have the correct firmware and hdlc embedded software loadded into the DAG3.7T card.(using dagrom, or the API).

3.5 Default behavior of the HDLC Embedded Software

After the XScale processor is running the DAG3.7T will start with out any filters enabled. And all traffic passing through.

Data is returned to the card in bursts. By default the burst size is one Mibyte, this can be changed by using the **aal_msg_set_burst_size()** function. Altering this size can make significant differences to the throughput of the HDLC filter, depending on the traffic, increasing the size introduces higher latency. When the burst size is changed, during filtering, the current packet of data will be sent at the *previous* bust size. All further bursts will be sent at the new size. If the traffic source is to be stopped (for example if the card is to be disconnected) then the **aal_msg_flush_burst_buffer()** function can be used to force any remaining part of the block to be sent to the host. This two functions are depreciated and may be not supported in future releases. The flushing will be done automatically if no new data for transmition is presented.

The software is expecting messages from the host for controlling functions(set up filters , read software id, ..) described later in this document. The delay for a one message (command) to take the appropriate action can be up to one packet processing from the whole message arrived into the HDLC module. The total delay depends on the host and traffic.

3.6 Statistics

The DAG 3.7T HDLC filterer can provide statistics on the number of packets dropped by the filters. The number of packets is only available on a per card basis on this card. Both values are available via the **dagsar_get_stats()** function, by giving different statistic requested values.

The values will be reset via the **dagsar_reset_stats_all()** function, and by the **dagsar_reset_stats()** function, when the particular statistic is specified. The **dagsar_reset_stats_all()** function should be called at the beginning of a user program to reset all counters. For further information on which statistics are currently available consult the function definitions.

3.7 Filtering

All filtering is performed on HDLC packets, prior to any other processing.

3.7.1 General Filtering

Bit masks can be applied to the HDLC header and depending if there is a match, the packet can then be discarded. The user has to provide two values to set the filter: a *bit mask* and a *match* value. To each ATM header (4 bytes, HEC is removed) the system calculates the logical AND with the *bit mask*. The result is compared with the *match* value. If they are equal, the result of the match is decided by the value of the *action* argument, which can take two values: *sar_accept* or *sar_reject*. If the values do not match, then the opposite result is taken. Note this filter is very simple and we advise to use the extended filtering

3.7.2 Extended Filtering

The extended filter module is able to perform simple comparison operations on any contiguous four bytes of data in the atm cell.

There are two types of filters available. The first is an operational filter which applies a mask to 4 bytes of data, and compares the result to an expected value using a defined filter comparison operator. This is similar to the general SAR filtering described above with the extensions of being able to use other logical operators, and being able to apply the filter operator

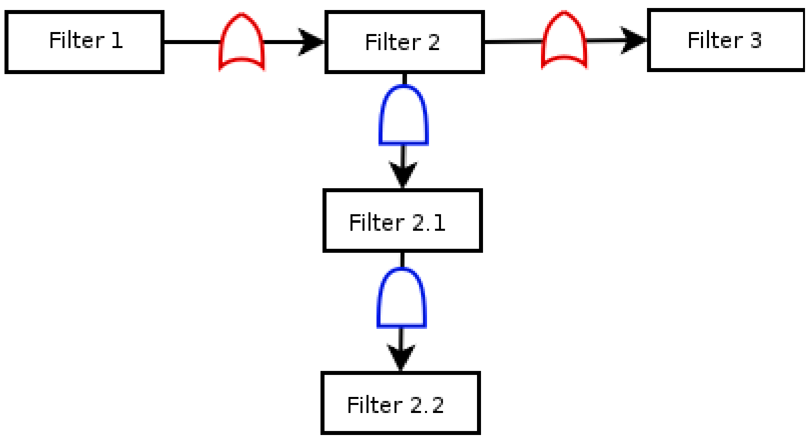
The logical operator is determined after the data and *bit mask* values have been logically AND'd together. In general filtering, this is a simple equal operator, where the result must be equal to the *match* value for the filter to be true. In extended filtering, this can also be: not equal, greater than, less than etc. For a full list of the available operators refer to the function definition.

The second type is a history filter, which applies a mask to 4 bytes of data (using an AND operator) and compares it to the masked value from the last time the filter run. If the result is a match, the filter is true and *action* is taken. If they do not match then the opposite to *action* is taken.

It is possible to have up to 32 different filters configured at any one time. This limit includes the one possible general filter entry.

3.7.3 Extended Filtering Examples

Extended Filtering also allows multiple filters to be linked together in a tree structure. This allows multiple levels of filters to be constructed by adding subfilters to filters. If for example, a filter structure is created with filters such as this:



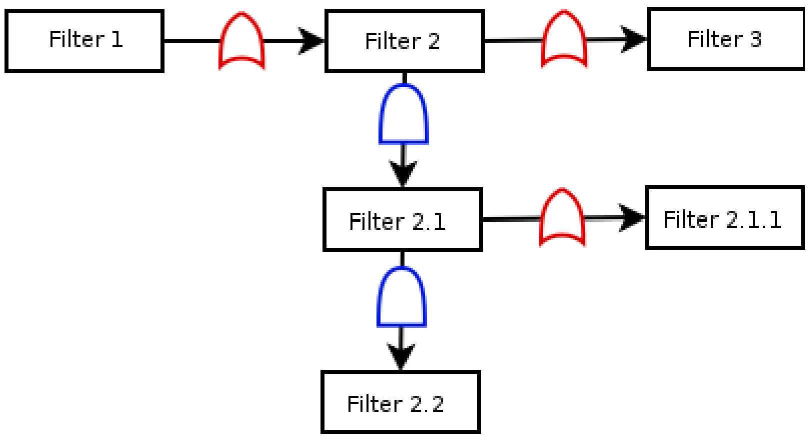
In the above example,

If Filter 1 is true, then *action* is taken.

If Filter 1 and 2 are false, there is no need to evaluate Subfilters 2.1 and 2.2 so Filter 3 is evaluated. If it is true *action* is taken, otherwise the opposite of *action* is taken.

If Filter 1 is false, and Filter 2 is true then Subfilters 2.1 and 2.2 are evaluated. If they are both true *action* is taken. If either or both of Filters 2.1 and 2.2 are false the opposite of *action* is taken.

To take this concept further, subfilters can be added to subfilters, for example:



With this example, in order for Subfilter 2.1 to be true, either of Subfilters 2.1 or 2.1.1 needs to be true and if both of them are false then the entire Filter 2 Subfilter tree is false and Filter 3 would then be applied.

Although this system allows a large amount of flexibility, it should be noted that adding many filters places additional load on the system. Therefore, it is not efficient to add many filters, unless doing so causes the majority of ATM cells to be dropped.

4. DAG EMA Library Function Definitions

4.1 dagema_reset_processor

```
int dagema_reset_processor
   int          dagfd
```

4.1.1 Description

The **dagema_reset_processor()** function resets and starts the XScale.

dagfd	The file descriptor for the DAG card as returned from dag_open().

4.1.2 Return Codes

Code	Description
0	Function was successful.
-1	Function was unsuccessful. Use the **dagema_get_last_error()** function for further error codes.

4.2 dagema_open_conn

```
int dagema_open_conn
   int          dagfd
```

4.2.1 Description

The **dagema_open_conn()** function opens the connection to the XScale, and starts the communications monitoring for the card. This function may only be called by a single application at a time.

dagfd	The file descriptor for the DAG card as returned from dag_open().

4.2.2 Return Codes

Code	Description
0	Function was successful.
-1	Function was unsuccessful. Use the **dagema_get_last_error()** function for further error codes.

4.3 dagema_close_conn

```
int dagema_close_conn
   int         dagfd
```

4.3.1 Description

The **dagema_close_conn()** function closes the communications with the XScale. This function should be called after all communications with the card are complete.

dagfd	The file descriptor for the DAG card as returned from dag_open().

4.3.2 Return Codes

Code	Description
0	Function was successful.
-1	Function was unsuccessful. Use the **dagema_get_last_error()** function for further error codes.

4.4 dagema_get_last_error

```
int dagema_get_last_error
   int         dagfd
```

4.4.1 Description

The **dagema_get_last_error()** function retrieves errno from the DAG ema library. This function can be used to retrieve further information from the library after a function call returns **-**1. Standard errno numbers are used to differentiate various errors.

dagfd	The file descriptor for the DAG card as returned from dag_open().

4.4.2 Return Codes

Code	Description
!0	Last value or errno

5. DAG 3.7T Specific Library Function Definitions

5.1 d37t_write_software_id

```
int d37t_write_software_id
   int          dagfd
   int32_t      num_bytes
   uint8_t      *datap
   uint32_t     key
```

5.1.1 Description

The **d37t_write_software_id()** function writes a new software ID into the EEPROM attached to the DAG3.7T's XScale processor. The key field is a simple security feature to prevent accidental access to the EEPROM; if the key does not match a specific value in the embedded software on the XScale then the XScale will freeze and a restart will be required.

The length of the ID must be at least 1 byte and no more than 128 bytes. (On the Rev B and Rev C DAG3.7T boards the EEPROM can store up to 128 bytes of data).

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
num_bytes	The number of bytes to write into the EEPROM. Between 1 and 128 inclusive. If the number is less than 128 then the remaining space is filled with 0.
datap	Points to a byte array containing the data to write to the EEPROM.
key	The write-enable key. This must be specified to enable write-access to the EEPROM. If the key is incorrect then EEPROM will not be written to and the XScale will lock up.

5.1.2 Return Codes

Code	Description
0	Function was successful.
-1	invalid number of bytes specified
-2	Firmware error writing to the EEPROM.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

5.2 d37t_read_software_id

```
int d37t_read_software_id
   int          dagfd
   int32_t      num_bytes
   uint8_t      *datap
```

5.2.1 Description

The **d37t_read_software_id()** function reads the software ID from the EEPROM attached to the DAG3.7T's XScale processor. The num_bytes field specifies how many bytes to read from the EEPROM. The contents are stored in the byte array pointed to by datap.

The length of the ID must be at least 1 byte and no more than 128 bytes. (On the Rev B and Rev C DAG3.7T boards the EEPROM can store up to 128 bytes of data).

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
num_bytes	The number of bytes to read from the EEPROM. Between 1 and 128 inclusive.
datap	Points to a byte array to be used to return the contents of the EEPROM.

5.2.2 Return Codes

Code	Description
0	Function was successful.
-1	invalid number of bytes specified
-2	Firmware error reading to the EEPROM.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

5.3 d37t_read_version

```
int d37t_read_version
   int         dagfd
   uint32_t    *version
   uint32_t    *type
```

5.3.1 Description

The **d37t_read_version()** function reads the version and type from the program running in the DAG3.7T's XScale processor.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
version	The version number of the program currently running on the scale processor.
type	Type of software running on the XScale processor. For the HDLC filtering this value will be equal to the defined variable HDLC (numerical value 3).

5.3.2 Return Codes

Code	Description
0	Function was successful.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

5.4 d37t_read_temperature

```
int d37t_read_temperature
   int         dagfd
   int32_t     sensor_id
   int         *temperature
```

5.4.1 Description

The **d37t_read_temperature()** function reads the current temperature from the LM63 temperature sensor device attached to the DAG3.7T's XScale processor.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
sensor_id	Specifies which temperature sensor to read. Rev B and Rev C DAG3.7T boards only have one so set this to 0. (0=default).
temperature	Points to a integer to store the temperature reading in.

5.4.2 Return Codes

Code	Description
0	Function was successful.
-1	Invalid sensor ID.
-2	Firmware error reading the temperature sensor.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

6. HDLC filtering Msg Function Definitions

Note: These API functions are relevant to the HDLC filtering. The functionality is similar to the AAL(ATM) filtering.

6.1 hdlc_msg_set_burst_size

```
int aal_msg_set_burst_size
   uint32_t    dagfd
   uint32_t    size
```

6.1.1 Description

The **aal_msg_set_burst_size()** is a DAG3.7T HDLC filtering module specific function that allows the amount of data that is written to the host at a time to be set. The burst size will not be altered in the block which is filling currently. All subsequent blocks will be at the size specified if the function is successful.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
size	The size of the burst to be sent in bytes. This can range between 72 and 10485760 (10MiB) and must be a multiple of 8 bytes.

6.1.2 Return Codes

Code	Description
0	Function was successful.
-2	Size argument was out of range.
-3	Timeout communicating with the XScale.
-4	Size argument was not a multiple of 8 bytes.
-6	Message not transmitted
-7	Message not responded to correctly

6.2 aal_msg_flush_burst_buffer

```
uint32_t aal_msg_flush_burst_buffer
   uint32_t    dagfd
```

6.2.1 Description

The **aal_msg_flush_burst_buffer()** function send any unfinished bursts of data to the host, regardless of the burst's size. This function can be used if the card is to be disconnected from the traffic source and all data is required at the host. This function should not be used when further data is expected. If the latency of the data is not suitable for an application the recommended solution is to reduce the burst size with the **aal_msg_set_burst_size()** function rather than using this function.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().

6.2.2 Return Codes

Code	Description
0	Function was successful.
-1	No data is available at the Reassembler to send
-2	No memory has been allocated to send
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

6.3 hdlc_msg_set_filter

```
uint32_t hdlc_msg_set_filter
   uint32_t    dagfd
   uint32_t    offset
   uint32_t    mask
   uint32_t    value
   uint32_t    operation
   uint32_t    filter_level
   uint32_t    level_conf
   uint32_t    history
   uint32_t    priority
```

6.3.1 Description

The hdlc**_msg_set_filter()** function sets a filter according to the settings given. For an overview of the filter operation, consult the overview section at the beginning of this document. A maximum of 256 filters and subfilters can be configured at any time.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
offset	The offset refers to the start location of the four bytes to be filtered on for this filter. The offset in measured in bytes and starts from the very beginning of the ERF that the ATM cell is contained in. Therefore an offset of zero would filter on the section of the timestamp of the ATM ERF. An offset of 20 corresponds to the ATM header. The offset must specify 4 bytes which will fall inside the ATM ERF, therefore no values greater than 68 will be valid.
mask	This is the 32 bit value used by the filter in the mask calculation. This involves performing a logical AND between this mask value and the data located in the ATM ERF at *offset.*
value	This is the value that the result of ANDing the data in the ATM ERF at *offset* with the *mask* value is compared to. This will determine whether the filter is true or false. In the case of a history filter this value is not used. Instead the value that was calculated in the previous application of this filter is used.
operation	This the operation used on an operational filter to compare the data at *offset* to the *value* argument. This filter can be equal, not equal, greater than or equal, less than or equal, greater than, less than, AND, OR or XOR. The enumeration filter_operations_t specifies the numerical values that correspond to these operations. The operation is not used on a history filter, as a history filter will always perform an equal operation between the data at *offset* ANDed with the *mask* and the previous data ANDed with the *mask*.
filter_level	A filter can be set at the Board level, where the filter will be applied to all data coming in to the board (DAG_FILTER_LEVEL_BOARD) , or the filter can be set at the connection level (DAG_FILTER_LEVEL_CHANNEL), where only data arriving with the specified connection number will be have the filter applied, or the filter can be at the Physical line level (DAG_FILTER_LEVEL_LINE), where only data arriving on the specified physical connection will have the filter applied. To specify which connection or line to filter on use the level_conf argument.
level_conf	This the the indication of which physical port (1-15) or connection (0-511) to filter on. This argument is only valid when the filter level is set to either DAG_FILTER_LEVEL_CHANNEL or DAG_FILTER_LEVEL_LINE during filter initialisation. At all other times this argument should be set to zero.

history	This is a boolean value which determines if this filter will be a history or an operational filter. A value of true (1) will cause this to become a history filter.
priority	The priority allows a heirachy of filters to be set up. When adding filters, they will be added in the priority order given. Therefore, to add a filter that should only be evaluated, if other filters have already been evaluated, add the filter with a lower priority number.

6.3.2 Return Codes

Code	Description
0	Function was unsuccessful.
!0	The unique filter identifier number. This can be used to reference the filter in future calls to delete the filter.
-2	The filter was unable to be allocated. This can be due to being out of memory or attempting to add more than 32 filters.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

6.4 hdlc_msg_set_subfilter

```
uint32_t hdlc_msg_set_subfilter
   uint32_t          dagfd
   uint32_t          offset
   uint32_t          mask
   uint32_t          value
   uint32_t          operation
   uint32_t          filter_level
   uint32_t          level_conf
   uint32_t          history
   uint32_t          priority
   uint32_t          parent_id
   list_operations_t list_operation
```

6.4.1 Description

The hdlc_**msg_set_subfilter()** function sets a filter according to the settings given, in a subfilter list below the parent filter specified. The subfilter list is created with the option of being an AND or OR list. For an overview of the subfilter operation, consult the overview section at the beginning of this document. A maximum of 32 filters and subfilters can be configured at any time.

dagfd	The file descriptor for the DAG card as returned from dag_open(). The card should have been initialized via dagema_open_conn().
offset	The offset refers to the start location of the four bytes to be filtered at. The offset is measured in bytes, and starts from the very beginning of the ERF that the ATM cell is contained in. Therefore, an offset of zero would start filtering at the ERF timestamp. An offset of 20 corresponds to the start of the ATM header. The offset must specify 4 bytes which will fall inside the ATM ERF, therefore no values greater than 68 will be valid.
mask	This is the 32 bit value used by the filter in the mask calculation. This involves performing a logical AND between this mask value and the data located in the ATM ERF at *offset.*
value	This is the value that is the result of AND'ing the data in the ATM ERF at *offset* with the *mask* value is compared to. This will determine whether the filter is true or false. In the case of a history filter this value is not used. Instead, the value that was calculated in the previous application of this filter is used.
operation	This the operation used on an operational filter to compare the data at *offset* to the *value* argument. This filter can be equal, not equal, greater than or equal, less than or equal, greater than, less than, AND, OR or XOR. The enumeration filter_operations_t specifies the numerical values that correspond to these operations. The operation is not used on a history filter, as a history filter will always perform an equal operation between the data at *offset* AND'd with the *mask* and the previous data AND'd with the *mask*.
filter_level	A filter can be set at the Board level, where the filter will be applied to all data coming in to the board (DAG_FILTER_LEVEL_BOARD), or the filter can be set at the connection level (DAG_FILTER_LEVEL_CHANNEL), where only data arriving with the specified connection number will be have the filter applied. To specify which level to filter on use the level_conf argument.
level_conf	This determines connection (0-511) to filter on. This argument is only valid when the filter level is set to DAG_FILTER_LEVEL_CHANNEL during filter initialisation. At all other times this argument should be set to zero.

history	This is a boolean value which determines if this filter will be a history or an operational filter. A value of true (1) will cause this to become a history filter.
priority	The priority allows a hierarchy of filters to be set up. When adding filters, they will be added in the priority order given. Therefore, to add a filter that should only be evaluated only when the other filters have already been evaluated, add the filter with a lower priority number then the other filters.
parent_id	This is the identifier of the filter that will be used as the starting point of this subfilter.
list_operation	The list_operation specifies if the filter to be created should be in an AND or OR list with the parent filter.

6.4.2 Return Codes

Code	Description
0	Function was unsuccessful.
!0	The unique filter identifier number. This can be used to reference the filter in future calls to delete the filter.
-2	The filter was unable to be allocated. This can be due to being out of memory or attempting to add more than 32 filters.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

6.5 hdlc_msg_set_filter_action

```
int hdlc_msg_set_filter_action
   uint32_t    dagfd
   uint32_t    action
```

6.5.1 Description

The hdlc_**msg_set_filter_action()** function sets the action that will be taken on HDLC packets which positively match any filters which are in place. Conversely the opposite of the action will be taken on any HDLC packets which do not match the filters. This is a global action that occurs on the results of all set filters.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
action	The action to be taken when an HDLC packet matches the filter value. The opposite action is performed on those HDLC packets which do not match the filter value after the bit mask has been applied.

The possible values for the action of the filter are:

sar_accept (0)	Any packets which are the same as the match value after processing the bitmask should be accepted. Any packets which are not the same as the match value after filter processing will be discarded immediately
sar_reject (1)	Any packets which are the same as the match value after processing the bitmask should be rejected. Any packets which are not the same as the match value after filter processing will be sent to the Memory hole for transferring to the HOST..

6.5.2 Return Codes

Code	Description
0	Function was successful.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

6.6 hdlc_msg_reset_filter

```
uint32_t hdlc_msg_reset_filter
   uint32_t    dagfd
   uint32_t    filter_id
```

6.6.1 Description

The hdlc_**msg_reset filter()** function allows a single filter to be deleted from the list of filters, without effecting any other filters. This function should not be used when attempting to delete a filter created with the more general **dagsar_set_filter_bitmask()** function.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
filter_id	The unique identifier of the filter to be deleted as returned by the hdlc**_msg_set_filter()** function.

6.6.2 Return Codes

Code	Description
0	Function was successful.
-2	Filter could not be deleted.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

6.7 hdlc_msg_reset_all_filters

```
int hdlc_msg_reset_all_filters
   uint32_t    dagfd
```

6.7.1 Description

The hdlc_**msg_reset_all_filters()** function removes all filters set, including those set with the more general **dagsar_set_filter_bitmask()** function.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().

6.7.2 Return Codes

Code	Description
0	Function was successful.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

7. DAG Sar Function Definitions

Note: All the functions are used the same way as the AAL reassembly. And shared on the host side the same api.

7.1 dagsar_get_stats

```
uint32_t dagsar_get_stats
      uint32_t    dagfd
      stats_t     statistic
```

7.1.1 Description

The **dagsar_get_stats()** function retrieves the internally held value that corresponds to the requested statistic. This is the total number of cells which fit the criteria for the statistic since the last restart or reset. The possible statistics currently available for the DAG3.7T are defined by the enumeration stats_t.

The arguments to the function are described below:

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
statistic	The statistic which should be returned

The currently available options for the statistic are:

dropped_cells	This value is not used in the HDLC filtering.
filtered_cells	This is the number of packets not returned to the host due to a filter that has determined the packet should be dropped.

7.1.2 Return Codes

Code	Description
Any	Value of the statistic

7.1.3 DAG3.7T Specific Return Codes

Code	Description
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

7.2 dagsar_get_interface_stats

```
uint32_t dagsar_get_interface_stats
      uint32_t    dagfd
      uint32_t    iface
      uint32_t    statistic
```

7.2.1 Description

The **dagsar_get_interface_stats()** function, when used on the DAG3.7T is functionally the same as the **dagsar_get_stats()** function, due to the DAG3.7T not having specified interfaces. This function retrieves the internally held value that corresponds to the requested statistic. This is the total number of cells which fit the criteria for the statistic since the last restart or reset. The possible statistics currently available for the DAG3.7T are defined by the enumeration stats_t.

The arguments to the function are described below:

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
iface	Specifies the interface. In the case of the DAG3.7T card, this should always be zero.
statistic	The statistic which should be returned

The currently available options for the statistic are:

dropped_cells	This is the number of cells not returned to the host, either due to the cells arriving on a Virtual Connection that is deactivated; or arriving on unconfigured Virtual Connections while in the Scanning mode.
filtered_cells	This is the number of cells not returned to the host due to a filter that has determined the cell should be rejected.

7.2.2 Return Codes

Code	Description
Any	Value of the statistic

7.2.3 DAG3.7T Specific Return Codes

Code	Description
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

7.3 dagsar_reset_stats

```
uint32_t dagsar_reset_stats
      uint32_t    dagfd
      stats_t     statistic
```

7.3.1 Description

The **dagsar_reset_stats()** function allows a single statistic to be reset to zero without affecting any other statistics, which will continue counting from their current position.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
statistic	The statistic which should be returned

The currently available options for the statistic are:

dropped_cells	This is the number of cells not returned to the host, either due to the cells arriving on a Virtual Connection that is deactivated; or arriving on unconfigured Virtual Connections while in the Scanning mode.
filtered_cells	This is the number of cells not returned to the host due to a filter that has determined the cell should be rejected.

7.3.2 Return Codes

Code	Description
0	Function was successful
!0	Function was unsuccessful

7.3.3 DAG3.7T Specific Return Codes

Code	Description
-2	Statistic is not recognised.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

7.4 dagsar_reset_stats_all

```
uint32_t dagsar_reset_stats
      uint32_t    dagfd
```

7.4.1 Description

The **dagsar_reset_stats_all()** function will reset all statistics to zero. It is recommended to call this function at the start of a program that will be using the statistics to put all statistics into a known state.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().

7.4.2 Return Codes

Code	Description
0	Function was successful
!0	Function was unsuccessful

7.4.3 DAG3.7T Specific Return Codes

Code	Description
-2	A Statistic was unable to be identified.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

7.5 dagsar_set_filter_bitmask

```
uint32_t dagsar_set_filter_bitmask
      uint32_t         dagfd
      uint32_t         iface
      uint32_t         bitmask
      uint32_t         match
      filter_action_t  filter_action
```

7.5.1 Description

The **dagsar_set_filter_bitmask()** function sets the values of the bitmask, match value and action to be taken for a filter on the DAG 3.7T card. These values define the filter on the DAG 3.7T. Any HDLC packets received will have the 32 bits of the HDLC packet header logically AND'd with the bitmask value supplied. The result of this calculation is then compared with the match value, if they are identical, the action defined be filter_action is then taken. The possible filter actions are:

sar_accept	Any ATM cells which are the same as the match value after processing the bitmask should be accepted. This will involve passing them onto the Virtual Connections list to determine what reassembly action should be taken. Any ATM cells which are not the same as the match value after filter processing will be discarded immediately
sar_reject	Any ATM cells which are the same as the match value after processing the bitmask should be rejected regardless of the Virtual Connection status of the ATM cell.. Any ATM cells which are not the same as the match value after filter processing will be processed by the reassembler normally.

The arguments to the **dagsar_set_filter_bitmask()** are:

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
iface	Specifies the interface. In the case of the DAG3.7T card, this should always be zero.
bitmask	This is the 32 bit value used by the filter in the mask calculation. This involves performing a logical AND between this value and the ATM header (32 bits not including the HEC).
match	The value which the calculation will need to match for the action specified by filter_action to occur. This argument is unused for the DAG3.7T card and should be left at the default.
filter_action	The action to be taken when an ATM cell matches the filter value. The contrary to this action is then performed on those ATM cells which do not match the filter value after the bit mask has been applied.

7.5.2 Examples

```
Action: ACCEPT
HDLC header:  11001011 11001000 11100111 00100010   0xCBC8E722
Bitmask:     11111111 00000000 00001111 00000000   0xFF000F00
-------------------------------------------------------------
Logical AND: 11001011 00000000 00000111 00000000   0xCB000700
Match value: 11110111 00000000 00000111 00000000   0xF7000700
(does not match -> packet rejected)

Action: REJECT
HDLC header:  11001011 11001000 11100111 00100010   0xCBC8E722
Bitmask:     11111111 00000000 00001111 00000000   0xFF000F00
-------------------------------------------------------------
Logical AND: 11001011 00000000 00000111 00000000   0xCB000700
Match value: 11110111 00000000 00000111 00000000   0xF7000700
(does not match -> packet accepted)

Action: ACCEPT
HDLC header:  11001011 11001000 11100111 00100010   0xCBC8E722
Bitmask:     11111111 00000000 00001111 00000000   0xFF000F00
-------------------------------------------------------------
Logical AND: 11001011 00000000 00000111 00000000   0xCB000700
Match value: 11001011 00000000 00000111 00000000   0xCB000700
(match -> packet accepted)

Action: REJECT
HDLC header:  11001011 11001000 11100111 00100010   0xCBC8E722
Bitmask:     11111111 00000000 00001111 00000000   0xFF000F00
-------------------------------------------------------------
Logical AND: 11001011 00000000 00000111 00000000   0xCB000700
Match value: 11001011 00000000 00000111 00000000   0xCB000700
(match -> packet rejected)
```

7.5.3 Return Codes

Code	Description
0	Function was successful
!0	Function was unsuccessful

7.5.4 DAG3.7T Specific Return Codes

Code	Description
-1	Filter could not be set
-2	The filter was unable to be allocated. This can be due to being out of memory or attempting to add more than 32 filters.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

7.6 dagsar_reset_filter_bitmask

```
uint32_t dagsar_reset_filter_bitmask
      uint32_t    dagfd
      uint32_t    iface
```

7.6.1 Description

The **dagsar_reset_filter_bitmask()** function resets the filter on the board so that no further packets are rejected based on the previous filter values.

dagfd	The file descriptor for the DAG card as returned from dag_open(). This card also should have been initialized via dagema_open_conn().
iface	Specifies the interface of the virtual connection to be activated. In the case of the DAG3.7T card, this should always be zero.

7.6.2 Return Codes

Code	Description
0	Function was successful
!0	Function was unsuccessful

7.6.3 DAG3.7T Specific Return Codes

Code	Description
-1	There is no identifiable filter on the card.
-2	Filter could not be deleted.
-3	Timeout communicating with the XScale.
-6	Message not transmitted
-7	Message not responded to correctly

8. Data Structures, Attributes, Defines and Enums

8.1 stats_t

The stats_t enumeration is defined in the dagsarapi.h file and is in the form

```
typedef enum
{
  dropped_cells,
  filtered_cells
}stats_t;
```

This defines the currently available statistics that can be received from the DAG3.7T HDLC filter. A definition of what these statistics represent can be found in the dagsar function definition section with the information for the function **dagsar_get_stats()**.

8.2 filter_action_t

The filter_action_t enumeration is defined in the dagsarapi.h file and is in the form

```
typedef enum
{
  sar_accept,
  sar_reject
}filter_action_t;
```

This defines the action which can be taken with a cell which is identified as fitting the set filter requirements. These actions can be set using the **dagsar_set_filter_bitmask()** function.

8.3 filter_operations_t

The filter_operations_t structure is defined in the aal_config_msg.h file and is in the form

```
typedef enum
{
    DAG_EQ = 0,      /* 0 equal */
    DAG_NEQ,         /* 1 not equal */
    DAG_LE,          /* 2 less than or equal */
    DAG_GE,          /* 3 greater than or equal */
    DAG_LT,          /* 4 less than */
    DAG_GT,          /* 5 greater than */
    DAG_AND,         /* 6 bitwise and */
    DAG_OR,          /* 7 bitwise or */
    DAG_XOR,         /* 8 bitwise exclusive-or */

    DAG_NUM_FILTER_OPERATIONS    /* this has to be the last in list */
}connection_info_t;
```

This defines the actions which can be used in extended filtering to compare the masked data value with the match value. This is only for an operational filter, the History filter always compares the masked data value with the previous masked data value, and if they are equal, then action is taken.

8.4 dag_filter_level_t

The dag_filter_level_t structure is defined in the aal_config_msg.h file and is in the form

```
typedef enum
{
    DAG_FILTER_LEVEL_BOARD = 0,
    DAG_FILTER_LEVEL_CHANNEL,

    DAG_NUM_FILTER_LEVELS   /* this has to be the last in list */

}DAG_filter_level_t;
```

This defines the level on which a filter is to be set.

8.5 list_operations_t

The list_operations_t structure is defined in the aal_config_msg.h file and is in the form

```
typedef enum
{
    DAG_OR_LIST = 0,
    DAG_AND_LIST

}list_operations_t;
```

This allows subfilter lists to be added in a form where either all of the filters need to be true to pass (DAG_AND_LIST) or any of the filters need to be true to pass (DAG_OR_LIST).

9. Data Formats

9.1 Generic Data Format

Data received from the DAG3.7T card HDLC filter is transmitted from the card in Extensible Record Format (ERF). The Generic ERF Format is as follows:

BYTE 3	BYTE 2	BYTE 1	BYTE 0
timestamp			
timestamp			
type	flags	rlen	
lctr		wlen	
(rlen - 16) bytes of record			

Timestamp	The time of arrival of this cell. Timestamps are in little-endian byte order (Pentium native). All other fields are big-endian byte order. No byte reordering is done on the Payload.
Type	This field contains an enumeration of the frame subtype. Valid types for the HDLC filter on the DAG3.7T card are: 5: TYPE_MC_HDLC
Flags	This byte is divided into 2 parts, the interface identifier, and the capture offset. 1-0: capture interface 0-3 2: varying record lengths present 3: truncated record [insufficient buffer space] 4: rx error [link error] 5: 5: ds error [internal error] 7-6: reserved
Rlen: Record Length	Total length of the record transferred over PCI bus to storage.
Lctr: Loss Counter	A 16 bit counter, recording the number of packets lost between the DAG card and the memory hole due to overloading on the PCI bus. The counter starts at zero, and sticks at 0xffff.
Wlen: Wire Length	Packet length including some protocol overhead. The exact interpretation of this quantity depends on the physical medium.

9.2 Multichannel HDLC ERF Record

The Multichannel HDLC ERF record is a fixed length record in the following format.

BYTE 3	**BYTE 2**	**BYTE 1**	**BYTE 0**
timestamp			
timestamp			
type: 5	flags	rlen	
lctr		wlen	
Multichannel Header			
HDLC Header(4 usually, but may vary depending the protocol)			
data(rlen –24) if hdlc is 4 bytes			

All fields are the same as the Generic Record Format unless listed below.

Flags	Capture interface is always zero. RX Error is set if any MC header Error bit is set.
Multichannel Header	This header is divided into several bit fields. Some of these fields are not used by the HDLC filter but are described here for continuity. 0-9 Connection number (0-1023) (512 connections are supported by DAG3.7T card) 10-15 Reserved 16-23 Reserved 24 FCS Error 25 Short Record Error (<5 Bytes) 26 Long Record Error (>2047 Bytes) 27 Aborted Frame Error 28 Octet Error. The closing flag wasn't octet aligned after bit unstuffing. 29 Lost Byte Error. The internal datapath had an unrecoverable error. 30 1st Rec. This is the first record received since this connection was configured. 31 Reserved
ATM Header	This does not include the 8-bit HEC

10. Examples

This section has sample code demonstrating the use of the API. Please note that all sections are written separately for clarity, but would not necessarily build or run individually.

10.1 Layer 2 filtering FISU's, LSSU's and MSU's filtering example

```
/* DAG headers. */
#include "dagsarapi.h"
#include "dagapi.h"
#include "dagutil.h"
#include "dagema.h"
#include "dag37t_api.h"
#include "d37t_i2c.h"

#include "dagaal/aal_config_msg.h"

#ifndef _WIN32
#define DEFAULT_DEVICE "/dev/dag0"
#else /* _WIN32 */
#define DEFAULT_DEVICE "dag0"
#endif /* _WIN32 */

static const char *const kDagaal5demoCvsHeader = "$Id: daghdlcdemo.c,v 1.1 2006/03/02 20:25:12 vladimir Exp $";
static const char *const kRevisionString = "$Revision: 1.1 $";

/* Command line configuration */
typedef struct
{
    int argc;
    char **argv;
    char *device;     /* Dag device */
    int start_xscale;  /* Reset and start xscale */
    int reset_all_filters;  /* Reset and start xscale */
    int use_drb_comm;  /* dag3.7t communication mode: drb or socket */

    uint32_t VCI;
    uint32_t VPI;
    uint32_t connectionNum;
    uint32_t max_size;
    uint32_t message;
    sar_mode_t sar_mode;
    net_mode_t net_mode;
    uint32_t swidkey;
} t_config;

t_config config;              /* Command line configuration */
int verbose = 0;              /* verbose output */
static char dagname[DAGNAME_BUFSIZE];

uint32_t filter_cnt;
uint32_t filters[256];

char *options =
    "daghdlcdemo is a demonstration of the Endace AAL reassembler.  For full\n"
    "details of the available functionality consult the SAR API guide.\n"
    "\n"
    "Usage: dagaal5demo [-d <dag>] [-x] \n"
    "    -d <device>         ; dag device to capture debug from [dag0]\n"
    "    [-x]                ; reset and start the xscale\n"
    "    -R                  ; reset all filters \n"
    "    -v -vv -vvv         ; verbose\n"
    "    --verbose           ; verbose (same as -v) \n"
    "   -h,--help,--usage    ; this page\n"
    "   -V,--version         ; display version information\n"
    "\n"
    "   List of message options\n"
    "   -C <connection num>  ; Connection Number to configure <16>\n"
    "   -s <max size>        ; AAL buffer size to use <65536>\n"
//  "   -m <sar mode>        ; reassembly type to use <2> 0=ATM 1=AAL2 2=AAL5\n"
    "   -n <net mode>        ; net type to use <0> 0=nni 1=uni\n"
    "   -k <swid key>        ; net type to use <0> 0=nni 1=uni\n"
    "                                                                              "
    "\n";


/*************************************************************************
* FUNCTION:     scanInputs(config, argc, argv)
* DESCRIPTION:  Read the command line options and fill in the config info.
* INPUTS:       config - pointer to the configuration info to fill in
*               argc   - number of command line tokens
*               argv   - command line token array
* OUTPUTS:      config populated with command line options
* RETURNS:      none
*************************************************************************/
void
scanInputs (t_config * config, int argc, char **argv)
{
    int opt = 0;
    int dagstream;

    config->argc = argc;
    config->argv = argv;
    config->device = DEFAULT_DEVICE;
    config->start_xscale = 0;
    config->VCI = 64;
    config->VPI = 28;
    config->connectionNum = 16;
    config->max_size = 1024*64;
    config->sar_mode = sar_aal5;
    config->net_mode = 0;

    while ((opt = getopt (argc, argv, "h?d:vxRV-:c:k:p:C:s:n:m:")) != EOF)
    {
        switch (opt)
        {
        case '?':
        case 'h':
            fprintf (stderr, options);
            exit (EXIT_SUCCESS);
            break;

        case 'v':
            verbose++;
            break;

        case 'x':
            config->start_xscale = 1;
            break;
        case 'R':
            config->reset_all_filters = 1;
            break;

        case 'd':
            /* Dag device */
            if (-1 == dag_parse_name(optarg, dagname , DAGNAME_BUFSIZE, &dagstream))
            {
```

```
            dagutil_panic("dag_parse_name(%s): %s\n", optarg, strerror(errno));
        }
        config->device = dagname;
        break;
    case 'k':
        /* Key for Software id change */
        sscanf(optarg,"%x",&config->swidkey);
        printf("Software ID KEY 0x%x\n",config->swidkey);
        break;

    case 'V':
        fprintf (stderr, "dagswid (DAG %s) %s\n", kDagReleaseVersion,
                 kRevisionString);
        exit (EXIT_SUCCESS);
        break;

    case '-':
        if (strcmp (optarg, "help") == 0 || strcmp (optarg, "usage") == 0)
        {
            fprintf (stderr, options);
            exit (EXIT_SUCCESS);
        }
        else if (strcmp (optarg, "version") == 0)
        {
            fprintf (stderr, "dagmap (DAG %s) %s\n", kDagReleaseVersion,
                     kRevisionString);
            exit (EXIT_SUCCESS);
        }
        else if (strcmp (optarg, "verbose") == 0)
        {
            verbose++;
            break;
        }
        else
        {
            fprintf (stderr, "unknown option '%c'\n", opt);
            exit (EXIT_FAILURE);
        }

    case 'c':  /* VCI to message*/
        config->VCI = strtol (optarg, NULL, 10);
        break;

    case 'p':  /* VPI to message*/
        config->VPI = strtol (optarg, NULL, 10);
        break;

    case 'C': /* Connection num to message*/
        config->connectionNum = strtol (optarg, NULL, 10);
        break;

    case 's':  /* max size to message*/
        config->max_size = strtol (optarg, NULL, 10);
        break;

    case 'n':  /* net mode */
        config->net_mode = strtol (optarg, NULL, 10);
        break;

    case 'm': /* sar mode */
        config->sar_mode = strtol (optarg, NULL, 10);
        break;

    default:
```

```
                fprintf (stderr, "unknown option '%c'\n", opt);
                exit (EXIT_FAILURE);
            }
        }
}

unsigned char soft_id[D37T_SOFTWARE_ID_SIZE+1];

int
main(int argc, char **argv)
{
        int res;
        int addr;
        int temperature;
        uint32_t version, type;
//      sar_mode_t mode = sar_aal0;
        int dagfd;


        printf("\nEndace DAG3.7T HDLC configuration demo\n");
        printf("(c) 2005 Endace Technology Ltd.\n\n");

        memset (&config, 0, sizeof (config));
        scanInputs (&config, argc, argv);


        if ((dagfd = dag_open(config.device)) < 0)
        {
                fprintf (stderr, "dag_open %s: %s\n", config.device,
                                 strerror (errno));
                exit(EXIT_FAILURE);
        }

        if (dag_set_mux(dagfd, DA_DATA_TO_LINE, DA_DATA_TO_IOP, DA_DATA_TO_HOST))
        {
                fprintf(stderr, "dag_set_mux failed\n");
                exit(EXIT_FAILURE);
        }

        /* Restart the xScale if required */
        if ( config.start_xscale )
        {
                printf("Restarting xScale ... please wait this may take up to 60 seconds to complete\n");
                if ( dagema_reset_processor(dagfd, 0) < 0 )
                {
                        printf("Failed to reset XScale (error code %d)\n", dagema_get_last_error());
                        exit(EXIT_FAILURE);
                }
        }


        /* Open a connection to the EMA */
        if ( (res = dagema_open_conn(dagfd)) < 0 )
        {
                printf("Failed to connect to board (error code %d)\n", dagema_get_last_error());
                exit(EXIT_FAILURE);
        }

        /* Set board debug flags */
        d37t_set_debug(dagfd, 0);
        //read inital software ID
        res = d37t_read_software_id(dagfd, D37T_SOFTWARE_ID_SIZE, soft_id);
        if (res)
        {
```

```
        printf("d37t_read_software_id() failed res=%d\n", res);
    }
    else
    {
        /* Replace any non-printable characters with spaces for this demo */
        for (addr=0; addr<D37T_SOFTWARE_ID_SIZE; addr++)
        {
            if (soft_id[addr])
            {
                if (!isgraph(soft_id[addr]))
                    soft_id[addr] = ' ';
            }
        }

        /* Null terminate it just in case */
        soft_id[D37T_SOFTWARE_ID_SIZE] = 0;

        printf("Software ID: \"%s\"\n", soft_id);
    }

    soft_id[10] = 0;
    soft_id[11] = 0;
    //write new software ID
    res = 0;
    if(config.swidkey)
        res = d37t_write_software_id(dagfd, 10, soft_id, config.swidkey);
        if (res)
        {
            printf("d37t_write_software_id() failed res=%d\n", res);
        };

    //red writen software id
    res = d37t_read_software_id(dagfd, D37T_SOFTWARE_ID_SIZE, soft_id);
    if (res)
    {
        printf("d37t_read_software_id() failed res=%d\n", res);
    }
    else
    {
        /* Replace any non-printable characters with spaces for this demo */
        for (addr=0; addr<D37T_SOFTWARE_ID_SIZE; addr++)
        {
            if (soft_id[addr])
            {
                if (!isgraph(soft_id[addr]))
                    soft_id[addr] = ' ';
            }
        }

        /* Null terminate it just in case */
        soft_id[D37T_SOFTWARE_ID_SIZE] = 0;

        printf("Software ID: \"%s\"\n", soft_id);
    }


    res = d37t_read_version(dagfd, &version, &type);
    if (res)
    {
        printf("d37t_read_version() failed res=%d\n", res);
    }
    else
    {
        printf("version = %u type = %u\n", version, type);
    }


    res = d37t_read_temperature(dagfd, 0, &temperature);
    if (res)
    {
        printf("d37t_read_temperature() failed, res=%d\n", res);
    }
    else
    {
        printf("Temperature: %d Degrees Celcius\n", temperature);
    }

//-------------------- Reset all Filters --------------------
    if(config.reset_all_filters) {

        printf("Reset all filters \n");
        res = aal_msg_reset_all_filters(dagfd);
        if (res<0)
        {
            printf("aal_msg_reset_all_filters() failed, res=%d\n", res);
        } else
        {
            filter_cnt = 0;
        };
    };

//-------------------- FISU Filter --------------------
    printf("Setup Filter for FISU packets fro MTP layer2\n");
    res = aal_msg_set_filter(dagfd,0,0x000000FC,0x00000000, DAG_EQ, DAG_FILTER_LEVEL_BOARD,0,0,0);
    if (res<0)
    {
        printf("aal_msg_set_filter() failed, res=%d\n", res);
    } else
    {
        filters[filter_cnt++]=res;
    };
//--------------------LSSU Filter --------------------------------
    printf("Setup Filter for LSSU packets\n");
    res = aal_msg_set_filter(dagfd,0,0x000000FC,0x00000004, DAG_GE, DAG_FILTER_LEVEL_BOARD,0,0,0);
    if (res<0)
    {
        printf("aal_msg_set_filter() failed, res=%d\n", res);
    } else
    {
        filters[filter_cnt++]=res;
    }
        // sub filter
    printf("Setup subfilter for LSSU packets\n");
    res = aal_msg_set_subfilter(dagfd,0,0x000000FC,0x00000008, DAG_LE, DAG_FILTER_LEVEL_BOARD,
                                        0,0,0,filters[filter_cnt-1],DAG_AND_LIST);
    if (res<0)
    {
        printf("aal_msg_set_subfilter() failed, res=%d\n", res);
    } else
    {
        filters[filter_cnt++]=res;
    };

//-------------------- MSU Filter--------------------------------------

    printf("Setup Filter for MSU packets\n");
    res = aal_msg_set_filter(dagfd,0,0x000000FC,0x0000000C, DAG_GE, DAG_FILTER_LEVEL_BOARD,0,0,0);
    if (res<0)
```

```
	{
		printf("aal_msg_set_filter() failed, res=%d\n", res);
	} else
	{
		filters[filter_cnt++]=res;
	}
//------------------------------------------------------------


	/* at this point we need to set up an aal5 connection on the required virtual connection*/


	/* the default in NNI so this is just done for demonstration purposes */
	//this is for AAL5 not fo hdlc
	//if (0 != dagsar_vci_set_net_mode(dagfd, DAG37T_INTERFACE, config.connectionNum,
	//	config.VPI, config.VCI, config.net_mode))
	//{
	//	printf("set net mode failed\n");
	//	return 1;
	//}
	//else
	//{
	//	printf("net mode set\n");
	//}

	/*set to reassemble AAL5 frames */
	//if (0 != dagsar_vci_set_sar_mode(dagfd, DAG37T_INTERFACE, config.connectionNum,
	//	config.VPI, config.VCI, config.sar_mode))
	//{
	//	printf("set sar mode failed\n");
	//	return 1;
	//}
	//else
	//{
	//	printf("sar mode set\n");
	//}

	//if (0 != dagsar_set_buffer_size(dagfd, config.max_size))
	//{
	//	printf("set buffer size failed \n");
	//	return 1;
	//}
	//else
	//{
	//	printf("buffer size altered \n");
	//}

	/* it is possible to check the mode a virtual connection is in.  This will verify the
	 * mode set earlier did set the mode to the correct option
	 */
	//mode = dagsar_vci_get_sar_mode(dagfd, DAG37T_INTERFACE,  config.connectionNum, config.VPI,
config.VCI);

	//if (mode != config.sar_mode)
//	{
//		printf("mode was not returned correctly %d \n", mode);
//		return 1;
//	}
//	else
//	{
//		printf("mode recognised\n");
//	}

	/* start the connection so data can be collected */
```

```
//    if (0 != dagsar_vci_activate(dagfd, DAG37T_INTERFACE,  config.connectionNum,
//          config.VPI, config.VCI))
//    {
//          printf("activation failed\n");
//          return 1;
//    }
//    else
//    {
//          printf("activation complete\n");
//    }


      dagema_close_conn(dagfd, 0);
      dag_close (dagfd);

      return EXIT_SUCCESS;
```

10.2 Set up and close down of card and communications

```
#include "dagapi.h"
#include "dagutil.h"
#include "dagsarapi.h"
#include "dag_ima_config_api.h"
#include "dagema.h"


int main(int argc, char **argv)
{
   int res;
   int dagfd;


   if ((dagfd = dag_open(config.device)) < 0) {
      fprintf (stderr, "dag_open %s: %s\n", config.device,
         strerror (errno));
      exit(0);
   }

   if (dag_set_mux(dagfd, DA_DATA_TO_LINE, DA_DATA_TO_IOP, DA_DATA_TO_HOST))
   {
      fprintf(stderr, "dag_set_mux failed\n");
      exit(0);
   }


   /* Restart the XScale if required */
   printf("Restarting XScale ...
            please wait this may take up to 60 seconds to complete\n");
   if ( dagema_reset_processor(dagfd, 0) < 0 )
   {
      printf("Failed to reset XScale (error code %d)\n",
         dagema_get_last_error());
      exit(0);

   }


   /* Open a connection to the EMA */
   if ( (res = dagema_open_conn(dagfd)) < 0 )
   {
      printf("Failed to connect to board (error code %d)\n",
          dagema_get_last_error());
      exit(0);
   }


   Communicate with the XScale or do some work here....
```

```
    /*Finished with the card now clean up */
    dagema_close_conn(dagfd, 0);
    dag_close (dagfd);

    return 0;
}
```

10.3 Get Information about card

```
Set up card and communications....


res = d37t_read_software_id(dagfd, D37T_SOFTWARE_ID_SIZE, soft_id);
if (res)
{
   printf("d37t_read_software_id() failed res=%d\n", res);
}
else
{
   /* Null terminate it just in case */
   soft_id[D37T_SOFTWARE_ID_SIZE] = 0;

   printf("Software ID: \"%s\"\n", soft_id);
}

res = d37t_read_version(dagfd, &version, &type);
if (res)
{
   printf("d37t_read_version() failed res=%d\n", res);
}
else
{
   printf("version = %u type = %u\n", version, type);
}


res = d37t_read_temperature(dagfd, 0, &temperature);
if (res)
{
   printf("d37t_read_temperature() failed, res=%d\n", res);
}
else
{
   printf("Temperature: %d Degrees Celcius\n", temperature);
}

Clean up the card and communications when finished.....
```

10.4 AAL Filtering if needed HDLC just replace the aal_ with hdlc_ in the filter setup

```
   Set up card and communications....


   /* Set up a simple filter via the dagsar api to remove all cells with VCI and
    * VPI of zero
    */

   if(0!= dagsar_set_filter_bitmask (dagfd, DAG37T_INTERFACE, 0xFFFFFFF0,
         0x00000000, sar_reject){
      printf("simple filter setting failed\n");
      return;
   }

   Wait for some data to be filtered....

   /*reset filter so data is no longer dropped*/
   if(0 != dagsar_reset_filter_bitmask(dagfd, DAG37T_INTERFACE)){
      printf("simple filter reset failed\n");
      return;
   }

   /* This time a more elaborate filter is to be added to demonstrate
    * the functionality available with the extended filtering module.
    */

   /* Any cells that do not fit the filter should be dropped */
   if(0 != aal_msg_set_filter_action(dagfd, sar_reject)){
      printf("action setting failed\n");
      return;
   }

   /*add a filter to remove all cells with connection number 128*/
   mainFilter = aal_msg_set_filter(dagfd, 20, 0x000000FF, 0x00000080, DAG_EQ,
         DAG_FILTER_LEVEL_BOARD, 0, false, 10);
   if(0 == mainFilter)
   {
      printf("setting main filter failed\n");
      return;
   }

   /* add a connection level filter on channel 16 that will remove all cells with
    * cids greater than zero*/
   connFilter = aal_msg_set_filter(dagfd, 20, 0xFF000000, 0x00000000, DAG_GE,
         DAG_FILTER_LEVEL_CHANNEL, 16, 0, 9);
   if(0 == connFilter)
   {
      printf("setting Connection filter failed \n");
      return;
   }

   /* add a subfilter from the board filter that will remove all cells with the
* first four data bytes all containing the value 0x6A in an AND sub filter     *
list */
   subFilter = aal_msg_set_subfilter(dagfd, 24, 0xFFFFFFFF, 0x6A6A6A6A, DAG_EQ,
         DAG_FILTER_LEVEL_BOARD, 0, false, 9, mainFilter, DAG_AND_LIST);
   if(0 == subFilter)
   {
      printf("setting subfilter failed \n");
      return;
   }

   Wait for the filter to process some data....

   /* remove the connection level filter */
   if(0 !=  aal_msg_reset_filter(dagfd, connFilter)){
      printf("single filter removal failed\n");
      return;
   }

   /* add another connection level filter on channel 17 that will remove all
    * cells with cids greater than zero*/
   connFilter = aal_msg_set_filter(dagfd, 20, 0xFF000000, 0x00000000, DAG_GE,
         DAG_FILTER_LEVEL_CHANNEL, 17, 0, 9);
   if(0 == connFilter)
   {
      printf("setting Connection filter on connec failed \n");
      return;
   }

   /*remove all filters*/
   if(0 != aal_msg_reset_all_filters(dagfd))
   {
      printf("failure when attempting to remove all filters\n");
      return;
   }


   Clean up the card and communications when finished.....
```

10.5 Statistics

```
Set up card and communications....

Set up some deactivated and filtered connections.....

/*reset all statistics in preparation */
if(0 != dagsar_reset_stats_all(dagfd)){
   printf("statistics reset failed\n");
   return;
}

wait for some time for statistics to be gathered on received data..

/*get statistics on dropped cells due to deactivated connections*/
NoOfDropped = dagsar_get_stats(dagfd, dropped_cells);

printf("%d Dropped Cells\n");

/*reset dropped cells statistic for new data */
if(0 != dagsar_reset_stats(dagfd, dropped_cells)){
   printf("dropped cells statistic reset failed\n");
   return;
}


/*get statistics on filtered cells*/
NoOfDropped = dagsar_get_stats(dagfd, filtered_cells);

printf("%d Filtered Cells\n");

/*reset filtered cells statistic for new data */
if(0 != dagsar_reset_stats(dagfd, filtered_cells)){
   printf("filtered cells statistic reset failed\n");
   return;
}

Clean up the card and communications when finished.....
```