endace
accelerated

# DAG Programming Guide

EDM04-19

### Protection Against Harmful Interference

When present on equipment this manual pertains to, the statement "This device complies with part 15 of the FCC rules" specifies the equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the Federal Communications Commission [FCC] Rules.

These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment.

This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications.

Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at their own expense.

### Extra Components and Materials

The product that this manual pertains to may include extra components and materials that are not essential to its basic operation, but are necessary to ensure compliance to the product standards required by the United States Federal Communications Commission, and the European EMC Directive. Modification or removal of these components and/or materials, is liable to cause non compliance to these standards, and in doing so invalidate the user's right to operate this equipment in a Class A industrial environment.

### Disclaimer

Whilst every effort has been made to ensure accuracy, neither Endace Technology Limited nor any employee of the company, shall be liable on any ground whatsoever to any party in respect of decisions or actions they may make as a result of using this information.

Endace Technology Limited has taken great effort to verify the accuracy of this manual, but nothing herein should be construed as a warranty and Endace shall not be liable for technical or editorial errors or omissions contained herein.

In accordance with the Endace Technology Limited policy of continuing development, the information contained herein is subject to change without notice.

### Website

http://www.endace.com

### Copyright 2008 Endace Technology Ltd. All rights reserved.

# Contents



 

 

# Introduction

## Overview

The DAG cards are a series of high performance PCI cards designed for ATM cell and packet capture on IP networks. Packet and cell transmission is also supported on some cards. Various versions have been produced supporting different physical layer interfaces. There are methods for accessing the DAG cards for both passive packet capture and for packet transmission.

## Purpose

The purpose of this DAG Programming Guide is to identify and explain:

- Collecting Network Data
- The C Application Program Interface
- Examples of Usage

## Related Documents

The following document provides additional information relating to ERF formats. This document is available on the Endace website at www.endace.com/support.

- *EDM11-01 ERF types*

 

## Providing network packet information

Collecting network data is based on providing network packet information and the DAG card functions. In passive capture mode all DAG cards reflect the arrival of a packet, or ATM cell, as a record within a large circular memory buffer provided by the PC host computer system.

An Extensible Record Format [ERF] record consists of a fixed size header that includes a high precision time stamp followed by data. This is a portion, or all, of the original packet present on the network link under observation and a small number of optional padding bytes to enforce alignment.

There are three methods of using DAG cards in order to provide network packet information to your application.

## Capture data

The first method is using the provided utility program `dagsnap` as a simple way to capture network data. `dagsnap` can write a file to disk containing the packet records in sequential order as presented by the DAG card. Such a trace file may later be processed by an analysis package.

If no output filename is specified, `dagsnap` will write to stdout, allowing the data to be piped into an analysis package that reads from its stdin. Although simple to prototype, this is not a high performance interface since all network trace data must pass through the UNIX pipe causing multiple memory copies and creating a bottleneck in the CPU-memory path.

PC-based computer systems can achieve between 400 – 1000 Mbytes/sec read/write performance when accessing non-cached data in main memory. When data is copied twice (or more) the effective throughput lowers to a bandwidth comparable to Gigabit Ethernet. DAGs can currently deliver up to 6.4 Gigabits per second peak data rate from a single DAG 6.1 OC-192c card to main memory.

## libpcap library

The libpcap library available from www.tcpdump.org now includes direct support for DAG cards. This allows any program written for the libpcap API to capture directly from DAG cards.

The DAG is not a NIC (network interface card), and much more efficient memory-mapped access is provided than is possible using NIC cards and drivers. This allows libpcap applications zero-copy access to packet headers and contents.

## Native C language

The third and most efficient method for retrieving network data is the native C language API. It provides the highest performance by providing a low-overhead zero-copy memory-mapped interface to the DAG. This API is described in detail in this manual.

# DAG card functionality

## Overview

DAG cards consist of:

- Line interface hardware
- A physical layer
- A packet processor that timestamps packet arrivals using the DAG Universal Clock Kit [DUCK] and creates the packet records
- An optional processing element
- The PCI Burst Manager [PBM] which writes packet records over the PCI bus into the host computer.

The physical layer interface is configured with an external tool, the `dagthree` program for the DAG 3.6 series for example. These tools report and set all physical layer attributes such as SONET scrambling and loopback. They are designed to be easily scriptable, and should be called at least once before the measurement software is run.
The packet processor controls such things as the ERF record length, and padding options. These packet processor options are also configurable with external tools.

## PCI burst m

The PCI burst manager does not require user configuration apart from controls to start and stop the measurement of network data and the allocation of memory between streams.
A large memory space is reserved per card at boot time by the `dagmem` driver. This memory space can then be divided into one or more large circular buffers, called streams. Receive streams use even identifiers, 0, 2, 4 for example, while transmit streams use odd identifiers 1, 3, 5 for example.
A particular firmware set for a DAG will have a maximum number of transmit and receive streams that it can support. The traditional receive-only DAG firmware application that is supplied with all DAG cards supports a single receive stream (0), and no transmit streams. Firmware that supports other stream configurations is available under separate licenses.

During receive operations the PCI burst manager receives packet records from the packet processor, and writes them into a receive stream buffer in host PC memory via Bus Mastering DMA. The PCI burst manager provides an indication to the driver of where it has filled the buffer to, the producer pointer, and will cease writing when the buffer is full. When the buffer becomes full, records are lost until the condition is cleared. Lost packets are counted and reported both in and out-of-band, described later.
The DAG API manages the consumer pointer, moving it forward as packet records are consumed by user programs in order to clear more space in the circular buffer. During transmit operations the user writes packets to be transmitted into a transmit stream buffer, which is also a large circular buffer in PC memory. The buffer pointers are then updated to reflect the new data available for transmission.
The PCI burst manager reads the data from the stream via Bus Mastering DMA and sends it to the physical layer for transmission

 

# C Application Programming Interface

## Overview

The programmer's interface to the DAG cards is exposed via a static or shared C library, libdag. Although it is possible to make operating system calls directly to the DAG device driver, this is discouraged as the kernel driver interface is subject to change without notice. The use of this library also aids portability.

Version 1.3 of the API is intended to provide data capture and transmission functionality. Functions include:

- Card open
- Buffer memory mapping
- Buffer pointer handling
- Packet record reception
- Packet record transmission
- Card close
- Functions not currently provided by the API include:
- Card reset
- Xilinx image loading
- Hardware configuration
- Status and statistics information

These functions may be included in future versions of the API, presently they should be addressed through the provided utilities, `dagreset`, `dagld`, `dagrom`, `dagthree`, `dagfour`, `dagsix`, `dagclock`. The public interface to `libdag` is defined in the header file `dagapi.h`.

## Receive model

API v1.1 and APIv1.2 supported DAG cards operating in a passive receive mode only. This capability was represented as a single receive-only data buffer.

API v1.3 introduces support for packet transmission, and the concept of multiple buffers per DAG card, called streams. Each stream consists of a buffer which can be used for either receiving or transmitting ERF packet records. Each stream can be addressed individually, with each receive stream presenting similar functionality to the traditional single buffer API.

Capture on each stream can be started and stopped independently, but the streams cannot be considered fully independent as some operations such as resets are available only globally. There are two access methods available for receive streams.

### dag_advance_stream()

The `dag_advance_stream()` function operates similarly to the `dag_offset()` function from API v1.2. It provides a pair of record aligned pointers into the stream buffer. The *bottom* pointer indicates where the next record begins.
The *top* pointer indicates the upper limit of available data. The user then steps through the records using the record length field from the ERF header until they reach the top pointer. They then pass in the top pointer as the new bottom pointer, freeing the processed space in the buffer. When `dag_advance_stream()` returns it will indicate the boundaries of a new block of data.
The `dag_advance_stream()` method is the most efficient interface, as it can pass large blocks of records to the user with a single call.
It does require the user to parse through the buffer content in order to process each record however. A simpler interface is also provided which is easier to use.

### dag_rx_stream_next_record()

The `dag_rx_stream_next_record()` function can be used in place of the `dag_advance_stream()` call. This function can simply be called repeatedly, each time returning a pointer to the beginning of the next available ERF packet record.
Buffer pointer management, freeing buffer space, and stepping through the available records are all implemented by the library. The disadvantage is that a function call on every packet is less efficient than the block oriented `dag_advance_stream()` method, with approximately 10% higher overhead.

### Version 1.4 of DAG API

Version 1.4 of the DAG API introduces only small changes from API v1.3, primarily to make the API more consistent by changing some error return codes. The only new function introduced is `dag_get_stream_buffer_level()` which can be used to report receive or transmit stream status.

### Version 1.6 of DAG API

Version 1.6 of the DAG API introduces one new function, `dag_rx_stream_next_inline()`, which is used for zero-copy inline forwarding of packets on transmit-capable DAG cards.

## Version numbering

The version numbering of the DAG API has changed to match the DAG software release number.

 

# Transmit model

The DAG API v1.3 introduced the capability to transmit packets, with appropriate firmware. Packet transmission firmware is not distributed as part of the standard DAG software package, and is available separately from Endace.

Packet transmission is accomplished via transmit streams, which can be considered similar to receive streams operating in reverse. Two access methods are provided for packet transmission.

## dag_tx_get_stream_space()

If the user application can construct its packet in a provided buffer, the user can call `dag_tx_get_stream_space()` to get a pointer into the stream buffer at which it can directly write one or more ERF records.

The user then calls `dag_tx_stream_commit_bytes()` to commit the appropriate number of bytes for transmission. This process is repeated to send further packets.

## dag_tx_stream_copy_bytes()

If the user wishes to transmit ERF packet records that are already present in memory in a user buffer, then `dag_tx_stream_copy_bytes()` is called.
This call will copy the ERF records from the user buffer into the DAG stream buffer and commit them for transmission. It is not necessary to call `dag_tx_get_stream_space()`. This method is less efficient as the ERF records must be copied at least once by the CPU from the user buffer to the stream buffer.

ERF records must be 64-bit aligned for transmission.

# dagapi h Header file

The C header file contains function prototypes for the DAG application programming interface [API].

## dag_open

### Purpose

The `dag_open` function is used in place of the system open function. The function prototype is:

```
int dag_open(char *dagname);
```

### Function

It is passed a string containing the DAG device node, “dag0”, for example and returns a valid UNIX file descriptor if successful.

If an error occurs it returns -1 with `errno` set as follows:

- ENOMEM (out of memory)
- ENFILE (too many open files for process)
- EIO (fatal internal error)
- ENODEV (the DAG does not support packet capture)

Error codes as for `ioctl(2), mmap(2).`

## dag_close

### Purpose

The `dag_close` close function terminates the access to a given DAG device represented by `dagfd`. The function prototype is:

```
int dag_close(int dagfd);
```

### Function

The function returns zero if successful, otherwise it returns -1 with `errno` set as follows:

- EBADF (dagfd is an invalid file descriptor)
- Error codes as for `close(2)`

## dag_configure

### Purpose

The `dag_configure` function allows user control over all parameters required to configure a card for a measurement session to be carried out. The function prototype is:

```
int dag_configure(int dagfd, char *params);
```

### Function

The `params` parameter is a string of configuration words separated by white space. Most configuration parameters will be identical to the ones used by the configuration tool. For instance, when operating a DAG 3.8 card, any command which is a valid option to `dagthree` will be valid.

### Options

At present, only the following options are implemented by `dag_configure`:

| dag_configure | Description |
|---|---|
| slen=<N> | Set packet capture length to N bytes. N will be rounded down to nearest multiple of four. Packets longer than N bytes will be truncated to N. |
| varlen | Set capture record mode to variable length. Packets shorter than slen will produce short records. |
| novarlen | Set capture record mode to fixed length. Packets shorter than slen will be padded to slen. |
| fixed | Same as novarlen. |
| ncells=<N> | Only for ATM capture mode on the DAG 3.5. N is 0-15, and specifies the number of cells to return from the start of each AAL5 frame. The DAG 3.5 is capable of tracking the state of many VPI/VCIs at once. If N is 0 all cells are captured, including OAM and RM cells which are dropped otherwise. |
| (no)lcells | Only for ATM capture mode on the DAG 3.5, where ncells>0. If set, the last cell of the AAL5 frame containing the AAL5 trailer will always be captured.<br>For example if ncells=3 and lcell is set, the first, second, third, and seventh cells will be captured from a seven cell AAL5 frame. |

The function returns zero if successful, otherwise it returns -1 with errno set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`params` contained an invalid token)
- EIO (fatal internal error)
- ENODEV (no such device)
- Error codes as for `lseek(2), read(2), fork(2), execvp(2), execlp(2), execve(2), wait(2)`

 

## dag_attach_stream

### Purpose

The `dag_attach_stream` function provides the user access to a particular receive or transmit data stream on a DAG card. The function prototype is:

```
int dag_attach_stream(int dagfd, int stream_num,
uint32_t flags, uint32_t extra_window_size);
```

### Function

It performs per-stream locking, and memory mapping functions. The DAG descriptor `dagfd` is provided by `dag_open()`. The `stream_num` is an integer in the range `0` to `MAX_INTERFACES` that indicates which stream to attach to. There are currently no flags defined.

To avoid seeing a packet record wrap over from the top of the circular buffer, a portion of the circular buffer is mapped into memory twice. This allows a record that would otherwise be split across the buffer boundary to appear contiguous. Traditionally the entire buffer is mapped twice.

For example, a physical 32MB buffer space would be mapped into user-space twice, consuming 64MB of that process's virtual memory. This permits operation of the circular buffer with no restriction on how often buffer space must be cleared. When `extra_window_size` has the special value 0, this behavior is maintained. This can consume significant quantities of process virtual memory however when large physical buffers are used, such as 512MB. The `extra_window_size` parameter can be used to specify the size of the second buffer mapping. For example, with a 32MB physical buffer and `extra_window_size` set to 4MB, only 36MB of process virtual memory is used.

When using the `dag_advance_stream()` access method if `extra_window_size` is non-zero, the user must never process more than `extra_window_size` before calling `dag_advance_stream()` again. This allows `dag_advance_stream()` to normalize the buffer pointers into the lower buffer mapping before the top of the second buffer mapping is reached.

When using the `dag_rx_stream_next_record()` access method, `extra_window_size` must be at least as large as the maximum size packet possible on the link medium. For efficiency `extra_window_size` should be at least several megabytes, 4MB is a reasonable default.

When attaching a stream for transmission, `extra_window_size` must be set equal to or greater than the maximum sized block of data that is to be transmitted at once. For backwards compatibility, or if a user is unsure, the `extra_window_size` is set to zero.

Setting the `extra_window_size` does not affect the physical memory consumption. Setting the `extra_window_size` only conserves the process virtual memory. Endace recommends setting the `extra_window_size` to zero unless process virtual memory is at a premium.

The function returns zero if successful, otherwise `MAP_FAILED` is returned with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EACCES (stream is locked by another process)
- EINVAL (`stream_num` is invalid or `extra_window_size` is too large)
- ENOMEM (stream has no memory allocated)
- ENODEV (the DAG does not support packet capture)
- EIO (fatal internal error)
- Error codes as for `ioctl(2), malloc(2), mmap(2)`

### Obsoletes

The `dag_attach_stream` function obsoletes `dag_mmap` (pg 19).

## dag_detach_stream

### Purpose

The `dag_detach_stream` function releases a stream on a DAG card. The function prototype is:

```
int dag_detach_stream(int dagfd, int stream_num)
```

### Function

The per-stream lock is released, allowing other processes to attach to the stream, and the stream buffer is unmapped from user space. It should be called when the stream is no longer required by the application.
The function returns zero if successful, otherwise it returns `MAP_FAILED` with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`stream_num` is invalid)
- Error codes as for `ioctl(2)`

## dag_start_stream

### Purpose

The `dag_start_stream` function starts a stream on a DAG card. The function prototype is

```
int dag_start_stream(int dagfd, int stream_num);
```

### Function

The stream must be attached before it can be started. The function returns zero if successful, otherwise it returns -1 with `errno` set as follows:

- EBADF (dagfd is an invalid file descriptor)
- EBUSY (the stream is already started)
- ENODEV (the DAG does not support packet capture)
- EINVAL (stream_num is not attached)
- EIO (fatal internal error)
- ETIMEDOUT (communication with card failed)
- Error codes as for ioctl(2)

### Obsoletes

The `dag_start_stream` function obsoletes `dag_start` (page 19).

## dag_stop_stream

### Purpose

The `dag_stop_stream` function stops the packet capture session. The function prototype is:

```
int dag_stop_stream(int dagfd, int stream_num);
```

### Function

The function returns zero if successful, otherwise it returns -1 with `errno` set as follows:

- EBADF (dagfd is an invalid file descriptor)
- EINVAL (stream_num is not attached or stream_num is not started)
- ENODEV (the DAG does not support packet capture)

### Obsoletes

The `dag_stop_stream` function obsoletes `dag_stop` (page 19).

 

## dag_get_stream_poll

### Purpose

The `dag_get_stream_poll` function is to read the polling parameters in use for a particular stream. The parameters are used in the function prototype:

```
int dag_get_stream_poll(int dagfd, int stream_num, uint32_t *mindata, struct
timeval *maxwait, struct timeval *poll);
```

### Function

The DAG drivers avoid interrupts due to the associated overheads, using polling methods instead. The amount of data that must be received before a call to `dag_advance_stream()` or `dag_rx_stream_next_record()` will return is given by `mindata`. This defaults to 16 bytes, the size of an ERF record header.

If `mindata` is zero, the receive functions will return immediately if no data is available, allowing non-blocking behavior.

The `maxwait` parameter is the maximum amount of time the receive functions should wait before returning. This overrides the `mindata` parameter, so that even if `mindata` is non-zero, the call will return with 0 bytes available after `maxwait` time. By default the `maxwait` parameter is set to the special value zero which means that it is disabled. This means that the receive calls will block indefinitely for `mindata` bytes.

If `mindata` bytes are not available when the receive function is first called, the library will sleep for `poll` time before checking for more data. This sleep avoids excessive polling traffic to the DAG card that may waste bus bandwidth, and frees the CPU for other processes.

Each time the library wakes from a poll sleep, the timeout as set by `maxwait` is checked, and the function will return if `maxwait` is exceeded. The default value of poll is 10ms, implying a maximum of 100 polls per second when no data is available. The value of poll should always be less than or equal to the value of `maxwait`, as the minimum sleep time is poll.

The poll sleep is implemented in user space using the `POSIX.1b nanosleep(2)` function. The current implementation of this call in Linux is based on the normal kernel timer mechanism, which has a resolution of 1/HZ, or 10ms on Linux/i386. This means that values of `maxwait` and `poll` less than 10ms will result in additional delay up to 10ms.

If the application uses a real time scheduler such as `SCHED_FIFO` or `SCHED_RR`, then sleep values up to 2ms will be performed as busy-waits. This allows for faster and more accurate polling, but will lead to high CPU utilization due to busy-waiting rather than releasing the CPU to the scheduler.

The function returns zero if successful, otherwise it returns -1 with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`stream_num` is not attached)

### Obsoletes

The `dag_get_stream_poll` function obsoletes `dag_get_pollparams` (page 20).

## dag_set_stream_poll

### Purpose

The `dag_set_stream_poll` function is used to configure the polling parameters for an individual stream when the defaults are not sufficient. The function prototype is:

```
int dag_set_stream_poll(int dagfd, int stream_num, uint32_t mindata, struct
timeval *maxwait, struct timeval *poll);
```

### Function

The `dag_set_stream_poll` parameters are as detailed for the `dag_get_stream_poll()` described in Topic 3.3.8 of this Chapter, the Section above.

All the parameters must be supplied in each call. The function returns zero if successful, otherwise it returns -1 with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`stream_num` is not attached)

### Obsoletes

The `dag_set_stream_poll` function obsoletes `dag_set_pollparams` (pg 20)

## dag_get_stream_buffer_size

### Purpose

The `dag_get_stream_buffer_size` function returns the size of the stream buffer in bytes if successful. The function prototype is:

```
int dag_get_stream_buffer_size(int dagfd, int stream_num);
```

### Function

The stream must be attached in order to determine the size of its buffer. On failure it returns -1 with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)

 

## dag_get_stream_buffer_level

### Purpose

The `dag_get_stream_buffer_level` function returns the number of bytes currently outstanding in the stream buffer. The function prototype is:

```
int dag_get_stream_buffer_level(int dagfd, int stream_num);
```

### Function

For transmit streams this is the number of bytes that have been committed by the user but have not yet been transmitted. Space allocated using `dag_tx_get_stream_space()` which has not been committed for transmission is not counted.

For receive streams this is the number of bytes of data available to the user for reading. This does include bytes that the user may have read but has not yet freed by calling `dag_advance_stream()` or `dag_rx_stream_next_record()`.

The `dag_rx_stream_next_record()` routine may not free buffer space occupied by previously read packets immediately for efficiency reasons.

The `dag_rx_stream_next_record()` call reads hardware registers on the DAG card, so each call will generate bus transactions. If polled at high rates this could potentially interfere with data capture or transmission operations.

On failure it returns -1 with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`stream_num` is not attached)

## dag_rx_get_stream_count

### Purpose

The `dag_rx_get_stream_count` function returns the number of receive streams supported by the DAG with the current firmware load. The function prototype is:

```
int dag_rx_get_stream_count(int dagfd);
```

### Function

This does not imply that all of the streams have memory allocated to their buffers. The DAG may support a greater or lesser number of streams with different firmware.

The function returns the number of receive streams if successful, otherwise it returns -1 with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)

## dag_tx_get_stream_count

### Purpose

The `dag_tx_get_stream_count` function returns the number of transmit streams supported by the DAG with the current firmware load. The function prototype is:

```
int dag_tx_get_stream_count(int dagfd);
```

### Function

This does not imply that all of the streams have memory allocated to their buffers. The DAG may support a greater or lesser number of streams with different firmware.

The function returns the number of transmit streams if successful, otherwise it returns -1 with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)

## dag_tx_get_stream_space

### Purpose

The `dag_tx_get_stream_space` function provides a pointer to size bytes of available space for the indicated stream. The function prototype is:

```
void *dag_tx_get_stream_space(int dagfd, int stream_num, uint32_t size);
```

### Function

It is necessary to acquire a pointer into the stream buffer at which to write the records to be transmitted. When packet transmission is being performed using the zero-copy, the `dag_tx_get_stream_space` function blocks the transmission until the requested space is available.

While polling for space to become available, it will sleep in `poll` time increments as set with `dag_set_stream_poll()`, freeing the CPU for other processes.

The function returns a pointer to the requested space if successful, otherwise it returns NULL with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- ENOTTY (`stream_num` is not a transmit stream)
- EINVAL (`stream_num` is not attached)
- ENOMEM (`stream_num` has no memory allocated)

 

## dag_tx_stream_commit_bytes

### Purpose

The `dag_tx_get_stream_commit_bytes` function provides a pointer to `size` bytes of available space for the indicated stream. The function prototype is:

```
void *dag_tx_stream_commit_bytes(int dagfd, int stream_num, uint32_t size);
```

### Function

In order to transmit data the first step for the user is to get a pointer to write to using `dag_tx_get_stream_space()`, then write their data at that location, and finally call `dag_tx_stream_commit_bytes()` to indicate that the data can be sent.

No pointer to the bytes to be sent is required, as the API holds this internally. The parameter `size` is the number of bytes that can be sent, this may be less than or equal to the size requested in the previous call to `dag_tx_get_stream_space()`, but must not be greater.

This function returns a pointer to the end of the transmitted block, but no data can be written at this location until `dag_tx_get_stream_space()` has been called again to ensure buffer space is available.

The function returns a pointer to the end of the transmitted block if successful, otherwise it returns NULL with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`stream_num` is not attached or `size` is larger than the stream buffer size)
- ENOMEM (`stream_num` has no memory allocated)
- ENOTTY (`stream_num` is not a transmit stream)

## dag_tx_stream-copy_bytes

### Purpose

The `dag_tx_get_stream_copy_bytes` function is used to transmit packets where the packet records are already present in a user buffer. The function prototype is:

```
int dag_tx_stream_copy_bytes(int dagfd, int stream_num, void * orig, uint32_t
size);
```

### Function

The records are copied from the user buffer into the stream buffer when space is available, and committed for transmission. No other functions need be called when using this method, but it is less efficient as the packet records must be copied by the CPU.

The pointer `orig` indicates the location of the user buffer to be copies, while `size` contains the number of bytes to be copied and sent. The buffer to be sent does not have to be record aligned, but if the buffer contains only the start of a packet record, that packet will not be transmitted from the DAG until the remainder of the record is supplied.

This call will block until space is available in the transmit stream buffer for all of the supplied data to be sent. The function returns the number of bytes successfully written if successful, otherwise it returns -1 with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`stream_num` is not attached or size is larger than the stream buffer size)
- ENOMEM (`stream_num` has no memory allocated)
- ENOTTY (`stream_num` is not a transmit stream)

## dag_rx_stream_next_record

### Purpose

The `dag_rx_stream_next_record` function is used for receiving ERF records individually, rather than the block oriented approach of `dag_advance_stream()`. The function prototype is:

```
void *dag_rx_stream_next_record(int dagfd, int stream_num);
```

### Function

This is a simpler approach and may ease porting, but due to the function call per packet the overhead may be 10% higher. The two methods should not be mixed on a single stream while the stream is started. The function uses the stream poll parameters described under `dag_get_stream_poll()`. These parameters define the blocking or non-blocking behavior, as well as the optional timeout and poll period.

If not configured with `dag_set_stream_poll()` the default stream parameters will cause `dag_rx_stream_next_record()` to block when no data is available.

The function returns a pointer to a single ERF record if successful (the ERF header contains an *rlen* field that specifies the size of the record). Otherwise it returns NULL with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`stream_num` is not attached or size is larger than the stream buffer size)
- ENOMEM (`stream_num` has no memory allocated)
- ENOTTY (`stream_num` is not a receive stream)
- EIO (ERF record has an invalid ERF type)
- EAGAIN (timeouts are enabled and a timeout occurs when no data is available)
- Error codes as for `dag_advance_stream` (page 6).

## dag_rx_stream_inline

### Purpose

The `dag_rx_stream_next_inline` function is used for processing packets inline. It is the inline version of `dag_rx_stream_next_record()`. The function prototype is:

```
void* dag_rx_stream_next_inline (int dagfd, int rx_stream_num, int tx_stream_num
```

### Function

The function can only be used with a DAG that is capable of transmitting packets and configured with the `overlap` option to set up the memory buffers for inline operation:

- For DAG 3 class cards: `dagthree -d dagN` default overlap
- For DAG 4 class series: `dagfour -d dagN` default overlap

The `dagfwddemo` program included in the `tools` directory of the DAG software release demonstrates the inline capabilities of a DAG card and serves as a fully-functional example for programmers.

`dagfwddemo` applies a user-defined BSD Packet Filter [BPF] expression to each packet received and only retransmits the packets that pass the filter.

 

Two streams are attached and started before this routine is used. One stream is for receive and the other for transmit. The main loop of an inline packet processing application using `dag_rx_stream_next_record()` will look similar to the following code:

```
while (keep_processing())
{
uint8_t* record = dag_rx_stream_next inline(uDagfd, RX_STREAM, TX_STREAM);
uint32_t bytes_to_commit;

process(record);
bytes_to_commit = ntohs(((dag_record_t *)record)
->rlen);

dag_tx_stream_commit_bytes(uDagfd, TX_STREAM,
bytes_to_commit);
}
```

#### Process Routine

The `process()` routine has up to three functions to perform:

| Function | Routine |
|---|---|
| Determine action for packet | Make application-specific determination about whether the packet is to be dropped or retransmitted. |
| Set packet to drop. | If the packet is to be dropped then the *rx error* bit in the ERF header flags byte must be set to 1. |
| Adjust *iface* field. | If the packet is to be transmitted out to the opposite interface from which it arrived, then the *iface* field in the ERF header flags must be adjusted. |

#### Interface Transmits

Some DAG card firmware has the capability to automatically rewrite the interface field in the ERF header so that packets received on interface 0 are transmitted via interface 1 and vice versa.

If the DAG card has been configured to rewrite the interface field then the software does not need to perform Step 3 described above.

#### Record Size

`dag_rx_stream_next_inline()` returns a pointer to a single ERF record if successful, the ERF header contains an *rlen* field that specifies the size of the record. Otherwise it returns NULL with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor.)
- ENOTTY (One of the stream numbers is invalid.)
- EINVAL (One of the streams is not attached.)
- EIO (The ERF record has an invalid ERF type. EIO is usually a fatal error and the capture session must be stopped.)

## dag_advance_stream

### Purpose

The `dag_advance_stream` function is used when receiving ERF record. The function prototype is:

```
void *dag_advance_stream(int dagfd, int stream_num, void **bottom);
```

### Function

Since it can return more than one record to the user at a time, it can be more efficient than using `dag_rx_stream_next_record()`. It operates by returning a pair of pointers into the stream buffer, which is mapped into the user process space in the `dag_attach_stream()` call. The bottom parameter is a pointer to a void pointer. On the first call the void pointer can be NULL. On subsequent calls, this should contain the address that the user has completed processing up to. The function can change the value of the bottom pointer to renormalize the circular buffer, so it is doubly referenced. The return value is a pointer to the top of the available buffer space. For example:

```
void *bottom=NULL, *top=NULL;
top = dag_advance_stream(dagfd, 0, &bottom);
```

Assuming the buffer is mapped into user space at 10000, `bottom` will now contain 10000, and if 10000 bytes were received `top` would contain 20000. Processing can now begin for ERF records, starting at `bottom` (10000) and continue until you reach `top` (20000).

If the first 5000 bytes are processed and it is then decided to call `dag_advance_stream()` again, the call would be:

```
bottom = bottom + 5000;
top = dag_advance_stream(dagfd, 0, &bottom);
```

After this call bottom may still contain 15000, but top may be 25000 if a further 5000 bytes were received while process the initial 5000 bytes are being processed. If the circular buffer needs to be normalized, then `bottom` can have a lower value after calling `dag_advance_stream()` than what was passed in. The process is always started from `bottom`. After calling `dag_advance_stream()` the `top` pointer will always have a higher value than the `bottom` pointer. Further example code is provided below.

The function uses the stream poll parameters described under `dag_get_stream_poll()`. These parameters define the blocking or non-blocking behavior, as well as the optional timeout and poll period. If not configured with `dag_set_stream_poll()` the default stream parameters will cause `dag_advance_stream()` to block when no data is available.

The function returns a pointer to the top of the available buffer space if successful, otherwise it returns NULL with `errno` set as follows:

- EBADF (`dagfd` is an invalid file descriptor)
- EINVAL (`stream_num` is not attached)
- ENOMEM (`stream_num` has no memory allocated)
- EIO (fatal internal error)

EIO is usually a fatal internal error and the capture session must be stopped and minimally restarted.

### Obsoletes

The `dag_advance_stream` function obsoletes `dag_offset` (page 20).

 

# Deprecated functions

The following functions are provided in `dagapi.h` for backwards code compatibility only and should not be used in new projects.

- dag_mmap
- dag_start
- dag_stop
- dag_offset
- dag_getpollparams
- dag_setpollparams

## dag_mmap

### Purpose

The dag_mmap function returns an address in user space which corresponds to the base of the circular packet buffer as utilized by the DAG card `dagfd`. The function prototype is:

```
void *dag_mmap(int dagfd);
```

### Function

This buffer pointer is used as the base address to locate valid network capture data as indicated by the offset pointer. On error the function will report `MAP_FAILED` with `errno` set accordingly.

### Obsoleted by:

The `dag_mmap` function is obsoleted by `dag_attach_stream` (page 9).

## dag_start

### Purpose

The `dag_start` function starts a measurement session on the nominated DAG. The function prototype is:

```
int dag_start(int dagfd);
```

### Function

The function returns -1 on error with an indication in `errno`, otherwise zero is returned.

### Obsoleted by:

The `dag_start` function is obsoleted by `dag_start_stream` (page 10).

## dag_stop

### Purpose

The `dag_stop` function stops a measurement session on the nominated DAG. The function prototype is:

```
int dag_stop(int dagfd);
```

### Obsoleted by:

`dag_stop_stream` (page 10).

## dag_offset

### Purpose

The dag_offset function returns the first address beyond the most recently written packet record in the circular buffer. The function prototype is:

```
int dag_offset(int dagfd, int *oldoffset, int flags);
```

### Function

oldoffset is the address of the previous offset as returned by the card or any other address the application wishes to be acknowledged as having completed the processing at.

The function hides the details of data wrapping across the end of the large memory buffer and updates the location pointed at by oldoffset.

All data between *oldoffset and the offset value as returned can be considered valid network measurement data.

The dag_nextpkt() function will eventually allow the processing of this window of data. At present the processing must be accomplished by user code.

The flags defined are:

- DAGF_NONBLOCK. This flag causes dag_offset to be non-blocking, otherwise the function blocks until at least one record is available.

### Obsoleted by:

The dag_offset function is obsoleted by dag_advance_stream (page 6).

## dag_get_pollparams

### Purpose

The dag_getpollparams function reads the polling parameters in use. The function prototype is:

```
void dag_getpollparams(int *mindatap, struct timeval *maxwait, struct timeval
*poll);
```

### Function

The parameters are used in dag_offset().

### Obsoleted by:

The dag_getpollparams function is obsoleted by dag_get_stream_poll (page 11).

## dag_set_pollparams

### Purpose

The dag_setpollparams function sets the polling parameters in use. The function prototype is:

```
void dag_setpollparams(int mindata, struct timeval *maxwait, struct timeval
*poll);
```

### Functions

The parameters are used in dag_offset().

### Obsoleted by:

The dag_setpollparams function is obsoleted by dag_set_stream_poll (page 12).

 

## Introduction

The functional examples for programming with the API library are based on the `dagsnap` or `dagbits` application programs for network monitoring, and `dagflood` for packet transmission.

## Single record receive

### Purpose

The `dag_rx_stream_next_record()` used to process captured network packets one at a time from stream 0.

### Example

In this example the `extra_window_size` parameter of `dag_attach_stream()` is set to 4MB to reduce the total amount of memory mapped to user space. The `dag_rx_stream_next_record()` function handles this internally.

```
/* open DAG, configure, and attach to stream 0 (receive) */
if((dagfd = dag_open(”/dev/dag0”)) < 0)
                panic("dag_open %s: %s\n", dagname, strerror(errno));

if(dag_configure(dagfd, ”slen=1536”) < 0)
                panic("dag_configure %s: %s\n", dagname, strerror(errno));

if(dag_attach_stream(dagfd, 0, 0, 4*1024*1024) < 0)
                panic("dag_attach %s: %s\n", dagname, strerror(errno));

if(dag_start_stream(dagfd, 0) < 0)
                panic("dag_start %s: %s\n", dagname, strerror(errno));
/* Initialise DAG Polling parameters. */
timerclear(&maxwait);
maxwait.tv_usec = 100 * 1000; /* 100ms timeout */
timerclear(&poll);
poll.tv_usec = 10 * 1000; /* 10ms poll interval */

/* 32kB minimum data to return */
dag_set_stream_poll(dagfd, 0, 32*1024, &maxwait, &poll);
while(run) {
                rec = (dag_record_t*)dag_rx_stream_next_record(dagfd, 0);
                if (rec) {
                        len = ntohs(rec->rlen);
                        /* User processing here */
                        process_packet(rec, len);
}               else { /* rec == NULL */
                if (errno != EAGAIN)
                        panic("dag_get_next_record: %s\n",
strerror(errno));
                }
}
/* finished; stop capture, detach from stream and close */
dag_stop_stream(dagfd, 0);
dag_detach_stream(dagfd, 0);
dag_close(dagfd);
```

# Multiple copy receive

## Purpose

Using `dag_advance_stream()` to process multiple captured network packets per call from stream 0.

## Example

In this example the `extra_window_size` parameter of `dag_attach_stream()` is set to 4MB to reduce the total amount of memory mapped to user space.

The user must not read more than 4MB of data before calling `dag_advance_stream()` again in the `dag_advance_stream()`. Reading more than 4MB of data may cause segmentation faults. An alternative is to set `extra_window_size` to zero, in which case all the data that `dag_advance_stream()` provides can be processed.

```
/* open DAG, configure, and attach to stream 0 (receive) */
if((dagfd = dag_open(”/dev/dag0”)) < 0)

if(dag_configure(dagfd, ”slen=1536”) < 0)
           panic("dag_configure %s: %s\n", dagname, strerror(errno));

if(dag_attach_stream(dagfd, 0, 0, 4*1024*1024) < 0)
           panic("dag_attach %s: %s\n", dagname, strerror(errno));

if(dag_start_stream(dagfd, 0) < 0)
           panic("dag_start %s: %s\n", dagname, strerror(errno));

/* Initialise DAG Polling parameters. */
timerclear(&maxwait);
maxwait.tv_usec = 100 * 1000; /* 100ms timeout */
timerclear(&poll);
poll.tv_usec = 10 * 1000; /* 10ms poll interval */

/* 32kB minimum data to return */
dag_set_stream_poll(dagfd, 0, 32*1024, &maxwait, &poll);


while(run) {
           processed = 0;
           if((top = dag_advance_stream(dagfd, 0, &bottom)) == NULL)
           panic("dag_advance_stream %s: %s\n", dagname,
strerror(errno));
           diff = top - bottom;
           if (diff == 0)
                        continue;

/* If more than say 4MB of data has been processed, then go back to main loop and call
dag_advance_stream again. This allows the space in the stream buffer occupied by that 4MB
of processed records to be released */
           while((run) &&
                        ((top-bottom)>dag_record_size) &&
                        ((processed+dag_record_size)<4*1024*1024)) {
                        rec = (dag_record_t*)bottom;
                        len = ntohs(rec->rlen);
```

 

```
/* break if the whole record is not available */
                if((top-bottom) < len)
                          break;
/* break if processing this record would go over our 4MB limit */
                if((processed+len)>4*1024*1024)
                          break;
                /* User processing here */
                process_packet(rec, len);

/* increment bottom pointer to next packet in block */
                bottom += len;

/* increment count of data processed since last dag_advance_stream() call */
processed += len;
                          }
                }
/* finished; stop capture, detach from stream and close */
dag_stop_stream(dagfd, 0);
dag_detach_stream(dagfd, 0);
dag_close(dagfd);
```

# Zero copy transmit

## Purpose

Using packet transmission allows users to request space in the transmit stream buffer, and then directly create their packet in the stream buffer memory, avoiding copies.

## Example

Another use would be reading one or more packet records from storage and writing them directly into the stream buffer, avoiding an intermediate copy.
When requesting buffer space the maximum is requested that is intended to be used. Committing to less than was requested is permitted.
In the following example it is assumed the user constructs a single ERF record. There is no requirement for the data supplied to be record aligned and the user may write more than one ERF record.

```
/* open DAG and attach to stream 1 (transmit) */
if((dagfd = dag_open(”/dev/dag0”)) < 0)
   panic("dag_open %s: %s\n", dagname, strerror(errno));

if(dag_attach_stream(dagfd, 1, 0, maximum_size) < 0)
   panic("dag_attach %s: %s\n", dagname, strerror(errno));

if(dag_start_stream(dagfd, 1) < 0)
   panic("dag_start %s: %s\n", dagname, strerror(errno));


while (run) {
   /* This will block until space is available */
   if((record=dag_tx_get_stream_space(dagfd, 1, maximum_size))
== NULL) {
panic("dag_tx_get_stream_space %s: %s\n", dagname,   strerror(errno));
   }

   /* user constructs packet here at *record */
   actual_size = construct_packet(record, maximum_size);
   if(dag_tx_stream_commit_bytes(dagfd, 1, actual_size) == NULL)
panic("dag_tx_stream_commit_bytes %s: %s\n", dagname, strerror(errno));
}
/* finished; stop stream, detach from stream and close */
dag_stop_stream(dagfd, 1);
dag_detach_stream(dagfd, 1);
dag_close(dagfd);
```

 

# Copy with transmit

## Purpose

The transmit with copy method can be used where the ERF packet record to be transmitted is already present in a user buffer and must be copied into the stream buffer for transmission.

## Function

In the following example it is assumed the user provides a buffer containing an ERF record.

There is no requirement for the buffer supplied to be record aligned and it may contain more than one ERF record.

```
/* open DAG and attach to stream 1 (transmit) */
if((dagfd = dag_open(”/dev/dag0”)) < 0)
            panic("dag_open %s: %s\n", dagname, strerror(errno));

if(dag_attach_stream(dagfd, 1, 0, maximum_size) < 0)
            panic("dag_attach %s: %s\n", dagname, strerror(errno));

if(dag_start_stream(dagfd, 1) < 0)
            panic("dag_start %s: %s\n", dagname, strerror(errno));

while (run) {
            /* user has ERF record at *record to transmit */
            record = acquire_packet_from_somewhere(&size);

            if(dag_tx_stream_copy_bytes(dagfd, 1, record, size) == NULL)
            panic("dag_ tx_stream_copy_bytes %s: %s\n", dagname,
            strerror(errno));
}

/* finished; stop stream, detach from stream and close */
dag_stop_stream(dagfd, 1);
dag_detach_stream(dagfd, 1);
dag_close(dagfd);
```

# Version History

| Version | Date | Reason |
|---|---|---|
| 1-11 | | Previous versions. |
| 12 | June 2006 | |
| 14 | September 2007 | New template and removal of some sections. |
| 15 | January 2008 | Added information re extra_window_asize segmentation fault. |
| 16 | November 2008 | Updated front matter. Minor formatting changes. Corrected use of `dag_tx_get_stream_space()`. Added details about version numbering change. |