Data plane library Reference Manual

NetLogic Microsystems

Oct 2009

Contents

Chapter 1

DP API Programming Model

INTRODUCTION

The data plane API (DP API, or just DP) is a runtime interface to a NETL7 device. This chapter contains an overview of the programming model of the DP.

STATUS

The DP indicates pass/fail status information via a value of the nlm_status enum type. More fundamentally, an nlm_status value gives a boolean error indication: Nonzero/true means that some error has occurred; zero/false means that no error has occurred. The true (error) values are distinguished by the positive values of the type, each of which is an element of the enum.

Each DP function returns an nlm_status value as its function value. Any struct that represents the result of a DP function indicates the function status by a element of the nlm_status type.

CONFIGURATION

The DP first defines a model of device configuration via the following configuration function:

```
err = nlm_estimate_device_config (n_flows, n_jobs, &conf)
```

such that

```
nlm_status err
uint32_t n_flows, n_jobs
struct nlm_device_config conf
```

The function nlm_estimate_device_config assigns values to the size members of its argument nlm_device_config struct, which sizes are the minimum required for an application of n_flows flows and n_jobs jobs. The returned estimate is for a single thread. When using multiple threads typically the user memory and system memory must be scaled by the number of threads. (The notions of flow and job are described in the SEARCH DATA section below.)

Configuration parameters (members of struct nlm_device_config)

The interaction between the device and the DP API is fundamentally similar across different devices. This calls for a common configuration struct. On the other hand, variations in device capabilities make certain aspects of the configuration device-dependent. The DP takes the approach of defining a single configuration struct that accommodates the configuration requirements of all devices (rather than defining a device-independent portion and a C union of separate device-dependent configuration structs). This section describes the configuration related to the common aspects of device-DP interaction. See the device-dependent sections of this document for the configuration model of each device.

The DP implementation is thread safe. Multiple threads can operate concurrently on the device.

```
uint32_t max_threads
uint32_t thread_id
uint32_t using_manager_thread
```

Dataplane thread specification.

The DP implementation supports a maximum of 256 threads to run concurrently on the device.

max_threads specifies the maximum number of dataplane threads that are to be run concurrently on the device. All resources are equally partitioned among the max_threads. thread_id identifies the thread to which this configuration structure belongs. Thread id zero is special and is the only thread that is allowed to perform certain operations described later in the document. max_threads should be set to a number of threads the user really intends to run, if not this will result in wasted resources.

Setting using manager thread allows running upto two threads to arbitrate and manage device input and output fifo's. This is in addition to the 256 threads the dataplane supports. The use of manager threads is optional, and can be beneficial when large number of dataplane threads are deployed.

The DP implementation uses two separate memory spaces for storage of internal data structures: system memory, memory pool. Both of these spaces are allocated by the application and the size of each space is determined by the application.

```
System memory is used for the interface between the DP software and
the device.  The device requires that the system memory be physically
contiguous.   The DP software requires that the system memory be
virtually contiguous.

The memory pool is used for DP software internal structures.  The DP
software requires the pool space to be virtually contiguous.  There are
no device interface structures within the memory pool, so there are no
issues of physical address or physical contiguity.

void *sysmem_base_virt
nlm_phys_addr sysmem_base_phys
uint32_t sysmem_size

    Application allocation of system memory.  sysmem_base_virt is the base
    virtual address of system memory.  sysmem_base_phys is the equivalent
    address in the physical address space.  sysmem_size is the total size
    of the system memory allocation.

uint32_t input_fifo_size
uint32_t output_fifo_size
uint32_t size_of_context_save_restore_area

    System memory configuration parameters.

    The device has a single input job fifo, and single output results
    fifo.  input_fifo_size is the number of elements in the input fifo,
    and output_fifo_size is the number of elements in the output fifo.

    As a search proceeds for a particular flow, the device generates
    internal data that characterize the state of the search.  The device
    uses an area within system memory for saving and restoring these
    internal search states.  This area is called the "context save/restore
    area".  size_of_context_save_restore_area is the number of bytes of
    system memory that are to be allocated to the context save/restore
    area.

void *memory_pool
uint32_t memory_pool_size
void *cookie
void *(*xmalloc) (void *cookie, uint32_t size)
void (*xfree) (void *cookie, void *ptr)

    Application allocation of memory pool.  memory_pool is the base
    (virtual) address of the memory pool.  memory_pool_size is the total
    size of the memory pool allocation. If memory_pool is proviced by
    the user then the pool is equally partitioned among the dataplane
    threads. Instead of providing a single pre-allocated memory pool
    the allocation routine function pointer xmalloc can be provided.
    Individual dataplane threads will then call this routine to allocate
    memory as required. At most memory_pool_size will be allocated
```

```
        by each dataplane thread. The function provided through xfree will
        be called to free up the resources allocated through xmalloc.

        Both memory_pool and xmalloc/xfree should not be specified at the
        same time.

        The specified cookie handle in the config file will be passed to
        the xmalloc and xfree calls as the first argument. The cookie
        value is not interpreted in any way by the runtime. The cookie
        can be different for each dataplane thread.

    nlm_phys_addr (*virt_to_phys) (void const *)
    nlm_phys_addr packet_base_phys
    void const *packet_base_virt

        The application passes search job data buffers into the DP as a virtual
        base address and a size.  But the search job is executed by the device.
        So the virtual address must be translated by the DP SW into a physical
        address, and the buffer must be physically contiguous.

        If the call-back pointer virt_to_phys is non-null, then it is called
        with argument equal to a job data buffer virtual address.  The
        function return value is the corresponding physical address.
        Otherwise, if virt_to_phys is null, packet_base_virt must be the
        base virtual address of the data area, and packet_base_phys must
        be the corresponding physical address.

    nlm_ring_id ring_id

        Ring number for this device - only has meaning on NLS2008. See discussion
              of Rings in the section below.

INITIALIZATION AND TERMINATION
------------------------------

The DP defines a model of device initialization and termination via the
following functions:

    err = nlm_device_init (&conf, &dev)
    err = nlm_device_fini (dev)

such that

    nlm_status err
    struct nlm_device_config const conf
    struct nlm_device *dev

The function nlm_device_init opens a DP device instance, initializes the
underlying device and returns a pointer to the nlm_device control struct that
represents the instance.  It allocates the control struct out of the memory
pool (conf.memory_pool/conf.xmalloc) that is passed to it. The conf configuration
```

parameters are copied into the control struct (so that the conf can be deallocated upon return from the initialization function).

For NLS2008 devices the ring_id config field must be 0.

The configuration struct nlm_device_config conf must be fully initialized before being passed to the initialization function. Beyond the minimum size, members that are returned by nlm_estimate_device_config, the values assigned to conf members depend on the requirements of the underlying device and on the resource requirements and capacities of the application. For device requirements, see the per-device sections of the present document (below).

The function nlm_device_fini terminates the (currently initialized) DP instance that is represented by its argument control struct. It closes an instance which has been opened by nlm_device_init. The nlm_device struct pointer is the same as that previously returned by nlm_device_init.

Only thread_id = 0 is allowed to make the above calls.

All other DP threads must first call the functions below to initialize themselves and get access to their respective device handle. The DP threads must use their own device handles when making any further DP API calls. When threads attach the conf structure must be an identical copy of the config structure used during nlm_device_init except for the thread_id.

```
   err = nlm_device_attach (&conf, &dev)
   err = nlm_device_detach (dev)
```

such that

```
    nlm_status err
    struct nlm_device_config const conf
    struct nlm_device *dev
```

Before the attach functions are called, nlm_device_init should have been successfully called by thread_id = 0 with ring_id = NLM_MASTER_RING.

In addition upto two manager threads per physical ring can attach to the device. When attaching the thread_id in the conf structure must be set to NLM_MANAGER_THREAD_ID.

NLS2008:

On NLS2008 devices, access to additional rings is provided using the above interfaces. For this the ring_id conf field must not be 0. First thread_id = 0 must attach then further threads can attach using the above API on the ring_id. Before the attach function can be called nlm_device_init() must have been successfully called with ring_id = 0 (and not yet had nlm_device_fini() called on it.). See Rings section for additional details.

```
SEARCH CONTROL
--------------

The DP organizes search control around the primitive concept of a rule.  What
constitutes a rule is not defined by the DP, that is, the concept of rule is
itself uninterpreted by the DP.  But the DP does specify the following: Rules
are collected into sets called "rule groups".  Rule groups are collected into
sets called "rule databases".  Within a single database, (1) each group is
uniquely designated by an unsigned integer called its "rule group identifier",
and (2) each rule within a single group is designated by an unsigned integer
called its "rule identifier".  The same rule identifier may be specified for
more than one rule within a single rule group.

What constitutes a rule is determined by the underlying device.  The device
is programmable in the sense that the device is able to load one or more
application-defined rule databases into the device.  The device defines
a range of unsigned integers (0, 1, 2, ..., MAXIMUM) called "database
identifiers" by which the application uniquely names a database at load time.
The MAXIMUM value of a database identifier is determined by the device.

The set of databases that are currently loaded are called the "currently
effective" databases.  Immediately after initialization (nlm_device_init),
there are no currently effective databases.  The DP defines functions whereby
the application can either load a database or unload a currently loaded
database.

    err = nlm_device_try_load_database (device, database, size, replacing_id, target_id,
                                        num_blocks)
    err = nlm_device_load_database (device, database, size, target_id, num_blocks)
    err = nlm_device_unload_database (device, target_id)

such that

    void const *database
    uint32_t size, replacing_id, target_id, num_blocks
    struct nlm_device *device

The APIs can only be invoked by nlm_device handle that was obtained by
device_init call with thread_id = 0.

The database argument is a valid database structure whose size in bytes is
given by the size argument.  Arguments replacing_id and target_id are HW
database identifiers.  Interpretation of num_blocks is device-dependent.

There are two kinds of loading model: a "stop scan" model, and a "nonstop scan"
model.  The two models constitute different uses of the DP by the application,
rather than different modes of operation that are intrinsic to the DP itself.
The difference amounts to just this: A stop scan model requires that some
database D1 (which may be currently loaded as replacing_id) be unloaded before
database D2 is loaded (as target_id).  In other words, searching with respect
```

to D1 (loaded as replacing_id) must be permanently terminated (permanently "stopped") before database D2 can be loaded (as target_id). The defining property of a nonstop model is the literal negation of the defining property of a stop model: A nonstop scan model always allows database D2 to be loaded without unloading any database. (Under a nonstop model, replacing_id is always -1, which is the DP's conventional way of denoting no database identifier.)

The terms "stop" and "nonstop" derive from a particular organization of the application, which organization anticipates field updates of successive versions of a single logical database such that the process of updating must tolerate the possibility of failure. For example, the application organizes its use of database identifiers in pairs so that at any point of time only half of the 2*N identifiers are actually in use (loaded). Then the application supports N logical databases in such a way that a newer version of the database that is loaded at identifier K can be loaded at target_id (!= K) and subsequent searches can be directed to target_id without requiring either unloading of K or waiting for all the K-flows to drain. This nonstop usage model protects (for example) against the circumstance that a field update of a database fails, having unloaded (without possibility of reloading) the previously successfully loaded earlier version.

The nlm_device_try_load_database is essentially a predicate that returns true (== zero, == NLM_OK) if and only if (1) the argument database structure is valid, (2) sufficient DP system memory resources exist to accommodate a load of this structure, and (3) sufficient device resources exist for the load. If replacing_id is not -1, then the resources currently allocated to replacing_id are counted toward the resources that will be available for target_id (stop scan model). The interpretation of num_blocks is device-dependent and contributes to the resource calculation.

SEARCH DATA

The DP is capable of simultaneously searching multiple independent data streams in parallel. These streams are called "flows". A flow is actually a device request queue. The queue elements (the device requests) are called "jobs". There are two kinds of job: A "data job" is a request to search a given data buffer relative to given rule group; a "control job" effects some change in the state of the flow (without any associated data search).

The flow itself can be of one of two types. The flow type determines the search extent for all data jobs that are enqueued to the flow: A "stateless flow" is such that each data job is searched independently of the data jobs that precede and succeed it in the flow. A "stateful flow" is such that the entire sequence of its data jobs is searched as a single sequence of bytes (the job boundaries being insignificant to the search).

When the application enqueues a job, it must supply a job identifier, called a "(job) cookie". The purpose of the cookie is to indicate the job to which a particular device result refers. The value of the cookie is completely determined by the application, but it must be sufficient by itself to allow

```
the application to uniquely identify the job across all the currently
enqueued jobs across all the currently open flows.

Flows are created and destroyed by the application:

    err = nlm_create_flow (device, flow_type, &flow)
    err = nlm_destroy_stateful_flow (device, flow, cookie)
    err = nlm_destroy_stateless_flow (device, flow)

such that

    struct nlm_device *device
    nlm_flow_type flow_type
    struct nlm_flow *flow
    void *cookie

The flow creation function (nlm_create_flow) constructs a flow of the specified
type (flow_type).  Before the flow can receive jobs, it should be bound to
at least one database-id, group-id pair. The maximum number of permissible pairs
is device dependent. The DP assumes that the database-id and group-id are
provided correctly by the application.

The type of the flow determines the function to be used to destroy it.
Destroying a flow reclaims all resources allocated to it.

Stateful flows have hardware contexts allocated to them - this requires
nlm_destroy_stateful_flow to create a special job to the hardware to
release these and hardware responds with an NLM_END_OF_JOB result
which is given back to the application. However, if a stateful flow
is created and no job ever submitted to the hardware using this flow,
then the call to nlm_destroy_stateful_flow reclaims software resources and
immediately returns NLM_END_OF_JOB. No special jobs are sent to the
hardware in this case.

For stateless flows, only software resources are reclaimed, which does
not require a special result.

Binding and unbinding search entities with a flow is achieved by:

    err = nlm_flow_add_database_and_group (device, flow, database_id, group_id);
    err = nlm_flow_remove_database_and_group (device, flow, database_id, group_id);

such that
    struct nlm_device *device
    struct nlm_flow *flow
    uint32_t database_id
    uint32_t group_id

The application can initialize a data job in an open flow, and it can retract
such an initialization:
```

```
    err = nlm_flow_enqueue_search (device, flow, start, end, cookie, &job)
    err = nlm_cancel_job (device, job)

such that

    struct nlm_device *device
    struct nlm_flow *flow
    void const *start, *end
    void *cookie
    struct nlm_job *job

The enqueue function does not actually submit the data job to the device.  It
initializes a data job structure for the given flow, data buffer, rule group
and job cookie.  Then it returns a pointer to a representing nlm_job struct.
The application eventually submits the job as a member of a list of initialized
jobs (each job of the list referenced by nlm_job struct pointer).

The data buffer is determined by the arguments start and end: Argument start
points to the first byte of the buffer; argument end points one byte above the
last byte of the buffer (so the buffer length equals the numeric value of the
end byte pointer minus the value of the start byte pointer).

The arguments start and end are virtual pointers.
HW can access any physical memory present on the system, so data buffer (packet)
does not need to be copied into a special location.
DP converts virtual addresses into physical addresses using the formula:
packet_start_phys = nlm_device_config->packet_base_phys
                    + start - nlm_device_config->packet_base_virt
or with
packet_start_phys = nlm_device_config->virt_to_phys (start)
callback if it is specified (non-zero).

In production system packet will get DMAed into some kernel buffer by NIC and DP
would only need to know the virtual and physical address of such kernel buffer
to process the packet, effectively doing zero-copy access.

There are DP resources attached to an initialized job (for example, the nlm_job
struct itself).  Therefore, the job must be either submitted or cancelled, in
order that these resources may be recovered.  The application may cancel any
initialized unsubmitted job; but cancellation is invalid after submission.

Data searching is effected by means of the following functions:

    err = nlm_start_jobs (device, job_count, jobs)
    err = nlm_get_all_search_results (device, max, results, &result_count)

such that

    struct nlm_device *device
    uint32_t job_count, max, result_count
    struct nlm_job *jobs[job_count]
```

```
    struct nlm_result results[max]
```

The start function (nlm_start_jobs) submits a list of initialized (but not yet submitted) jobs to the device. The device begins searching these jobs and generating match reports. The order of searching jobs is preserved within a single flow, but the order of searching jobs across flows is undefined. A job that has been started can no longer be cancelled (nlm_cancel_job).

The probe function (nlm_get_all_search_results) is the means whereby the application fetches the results that are generated by the device. The function polls the DP for device results; it does not block waiting for results. In particular, if there are no unfetched results, then the function immediately returns result_count zero (without having modified any element of the results array). Otherwise, if there are results, then the probe function returns a positive (never greater than max) value of result_count, and gives the results as results[0..result_count-1] (again without having modified any other elements of the results array). The application must then analyze the sequence of results that has been returned to it.

Each result is of the following form:

```
    struct nlm_result
    {
      nlm_status status;
      void *cookie;
      union
      {
        struct
        {
          uint32_t rule_id;
          uint32_t group_id;
          uint32_t database_id;
          uint32_t byte_offset;
          uint64_t flow_offset;
          uint32_t match_length;
        } match;
        struct
        {
          uint64_t timestamp;
          uint32_t total_state_cnt;
          uint8_t peak_state_cnt;
          uint8_t final_state_cnt;
        } stats;
      } u;
    };
```

Let R be such a struct, as returned by the probe function. The value R.status determines the interpretation of the rest of the struct:

[1] If R.status has value zero (no error, NLM_OK), then R reports a data job match. R.cookie identifies the job in which the match has occurred. The

```
    members of R.u.match give the match data:

    [a] match.rule_id is the identifier of the rule that has triggered
        the match.

    [b] match.byte_offset is the (byte) offset from the beginning of
        the job of the last byte of the match.

    [c] In a stateless flow, match.flow_offset is identical to match.byte_offset.
        In a stateful flow, match.flow_offset is the data byte offset from the
        beginning of the flow of the last data byte of the match.

        The interpretation of offset is device-dependent.  There are two
        alternative interpretations of offset: Either (i) the offsets are data
        byte offsets (the first data byte being at offset zero), or (ii)
        offsets indicate positions preceding each data byte in the flow.  On
        interpretation (i), a match before the first character is at offset
        (uint64_t)-1; while on interpretation (ii), a match before the first
        character is at offset 0.

    [d] match.match_length is the total (data byte) length of the match
        (in the flow).

        The interpretation of match length, and even the existence of a well
        defined match length, is device-dependent.  The match length always
        indicates the last data byte of the match, but the number of matches
        that may be reported (for example, as a result of multiple rules
        triggering a match at the same position) varies from device to device.
        Some devices do not report a match length at all.

    Relative to a single flow, the sequence results[0..result_count-1] is in
    nondecreasing order of offset. (The order is not necessarily strictly
    increasing, since two rules can trigger matches at identical offsets.)

    In a stateful flow, it is logically possible for the 64 bit flow-relative
    flow_offset to overflow.  If it does overflow, then it numerically wraps.
    There is no other indication of the overflow, since this is a natural, not
    an error condition.  The application has the responsibility to anticipate
    (or preclude) the possibility of flow_offset overflow, and to correctly
    handle the overflow if it does occur. (Overflow of the job-relative
    byte_offset is logically impossible in the current 32 bit implementation,
    because the total job length is always the difference of two 32 bit
    pointers.  And, for the same reason, it is logically impossible for the
    flow_offset to overflow in a stateless flow.)

[2] If R.status has (positive error) value NLM_END_OF_JOB, then R marks
    the end of the processing of the job identified by R.cookie.  R.u.stats
    gives information about the completed job.

[3] Otherwise, if R.status is neither zero nor NLM_END_OF_JOB, then
    the job has been terminated abnormally and R.status is the (positive)
```

```
    NLM_DEVICE_* error code that indicates the cause of the termination.

When multiple DP threads are deployed, each thread must use its own device
handle obtained by call to nlm_device_init/nlm_device_attach to call the
above API's. Flows, jobs created on one device handle should not be passed
to another.

DEVICE PARAMETERS
-----------------

During the running of the system (between nlm_device_init and nlm_device_fini),
the application has need to inquire and (in some cases) modify certain device
parameters.

The application accesses device parameters via the following two calls:

    err = nlm_device_get_param (device, param, ...)
    err = nlm_device_set_param (device, param, ...)

such that

    enum nlm_device_param param
    struct nlm_device *device
    nlm_status err

The APIs can only be invoked by nlm_device handle that was obtained by
device_init call with thread_id = 0.

The interpretation of a device parameter is necessarily device-dependent, but
the DP takes the approach of defining a single set of parameters (and for
each parameter a fixed sequence of argument types) that is a superset of the
parameters that are interpreted by any single device.  If a device has no
interpretation of a particular parameter, then it returns status NLM_UNSUPPORTED.
This is not really an error, since it represents the natural circumstance that
not all parameters have an interpretation on all devices. Members of the
nlm_device_param enumeration can be classified into different logical groups.
Elements of one such group configure the device, those from another
group read auxiliary information from the device while elements
from yet another group control the behavior of the loader. Based on its
use, each element may be get/set/both.

The members of the nlm_device_param enum
(together with each member's vararg argument sequence) are as follows:

    NLM_DATABASE_BLOCK_CNT (get/set)

        vararg[0] database_id: uint32_t
        vararg[1] num_blocks: (get) pointer to uint32_t; (set) uint32_t

        Get: Inquire the number of HW database blocks that are currently
        allocated to database_id.
```

```
    Set: Allocate num_blocks HW database blocks to database_id.  It is
    required that database_id be loaded.  The number of blocks that are
    actually allocated is implicitly constrained (by the DP) to a maximum
    of the total number of blocks of the database that are currently
    loaded at database_id.  For load policy zero (implicit/automatic
    policy), num_blocks zero is interpreted to mean all available blocks
    (up to the total number of blocks of the database).

NLM_DATABASE_TOTAL_BLOCK_CNT (get only)

    vararg[0] uint32_t database_id
    vararg[1] blocks: pointer to uint32_t

    Inquire the total number of database blocks (HW blocks and
    system memory extension area blocks) that are currently loaded
    for database_id.  This is the compiled size of the database.

NLM_DATABASE_LOAD_POLICY (get/set)

    vararg[0] policy: (get) pointer to uint32_t; (set) uint32_t

    The load policy determines the method by which HW blocks are allocated
    to loaded databases.  The load policy affects the function of the
    following operations: nlm_device_try_load_database, nlm_device_load_-
    database, nlm_device_unload_database, nlm_device_set_param (parameter
    NLM_DATABASE_BLOCK_CNT).

    There are two load policies:  Load policy zero is the "implicit" or
    "automatic" policy.  Load policy one is the "explicit" policy.

    If load policy zero is in effect, the DP implementation balances
    the distribution of HW blocks across the currently loaded database
    identifiers. If load policy one is in effect, the HW block
    distribution is entirely defined by the application.

NLM_DATABASE_COPY_CNT (get/set)

    vararg[0] database_id: uint32_t
    vararg[1] num_copies: (get) pointer to uint32_t; (set) uint32_t

    Get: Returns the number of instances of the database in the device
    memory.

    Set: Triggers the loader to grow/shrink the number of instances
    of the given database in device memory.

NLM_DATABASE_LOADER_STATUS (get)

    vararg[0] status : (get) pointer to nlm_status;
```

```
    Function to poll if the loader is done with the most recent
    nlm_device_load_database (or, nlm_device_try_load_database) call.

NLM_TOTAL_BLOCK_CNT (get only)

    vararg[0] blocks: pointer to uint32_t

    The size of HW database memory, in blocks.

NLM_FLOW_OFFSET (get/set)

    vararg[0] flow: pointer to struct nlm_flow
    vararg[1] offset: (get) pointer to uint64_t; (set) uint64_t

    Inquire/define the flow offset for the specified flow.  This offset
    is effective for the next job started in the flow.

NLM_INPUT_FIFO_TIMER (get/set)

    vararg[0] curval: (get) pointer to uint32_t; (set) uint32_t

    The rate at which the device examines the read pointer of its single
    job input fifo.

NLM_OUTPUT_FIFO_TIMER (get/set)

    vararg[0] curval: (get) pointer to uint32_t; (set) uint32_t

    The rate at which the device updates its fifo pointers (both the result
    fifo pointer and the input fifo pointer are updated atomically).

NLM_MAX_MATCHES_PER_JOB (get/set)

    vararg[0] curval: (get) pointer to uint32_t; (set) uint32_t

    The maximum number of matches for a single search job.

NLM_MAX_STATES_PER_BYTE (get/set)

    vararg[0] curval: (get) pointer to uint32_t; (set) uint32_t

    The maximum number of internal states per byte for the given
    traffic and rule database. Higher states per byte can adversely
    impact performance.

NLM_HW_REGISTER  (get/set)

    vararg[0] reg_addr : uint32_t
    vararg[1] reg_val  : (get) pointer to uint32_t; (set) uint32_t

    Read/write the given HW register. Refer device data-sheet
```

```
        for list and description of supported registers.

    NLM_TIMESTAMP (get only)

        vararg[0] timestamp: pointer to uint64_t

        Read the device HW time counter.

    NLM_PACKET_ENQUEUE_POLICY (get/set)

        vararg[0] policy: (get) pointer to uint32_t; (set) uint32_t

NLS025/NLS055 DEVICES (mars1)
---------------------

Configuration parameters (struct nlm_device_config members)

    The configuration model of the NLS105/NLS205 devices is identical to the
    generic configuration model described earlier. Additionally, the following
    attribute of nlm_device_config is applicable for these devices as
    described below.

    uint32_t size_of_database_extension_area

        A device database load, distributes the database definition across
        both device HW memory and system memory.  The area of system memory
        that is dedicated to database definition is called the "database
        extension area".  size_of_database_extension_area is the number of
        bytes of system memory that are to be allocated to the database
        extension area.

Device parameters (nlm_device_get/set_param arguments)

    The size of one database block is 8192 entities, where the size of an entry
    is 12 bytes.  (One database block is 8192 * 12 == 98304 bytes.)  There are
    exactly 16 HW database blocks.

    The following table indicates the device parameters that are applicable
    for this family of devices. ('-' means NLM_UNSUPPORTED is returned).

                                         get         set
    NLM_FLOW_OFFSET                       y           y
    NLM_HW_REGISTER                       y           y
    NLM_MAX_MATCHES_PER_JOB               y           y
    NLM_MAX_STATES_PER_BYTE               y           y
    NLM_OUTPUT_FIFO_TIMER                 y           y
    NLM_INPUT_FIFO_TIMER                  y           y
    NLM_TIMESTAMP                         y           -
    NLM_DATABASE_LOAD_POLICY              y           y
    NLM_DATABASE_BLOCK_CNT                y           -
    NLM_DATABASE_TOTAL_BLOCK_CNT          y           -
```

```
    NLM_TOTAL_BLOCK_CNT                  y          -
    NLM_DATABASE_COPY_CNT                -          -
    NLM_DATABASE_LOADER_STATUS           -          -
    NLM_PACKET_ENQUEUE_POLICY            -          -

Database load model.

    nlm_device_try_load_database (device, database, size, replacing_id, target_id,
                                  num_blocks)
    nlm_device_load_database (device, database, size, target_id, num_blocks)
    nlm_device_unload_database (device, target_id)

    There are exactly two database identifiers (0 and 1).

    replacing_id must be either identical to target_id or -1 (no identifier).

    num_blocks specifies a number of HW blocks.  num_blocks is either zero or a
    HW block count.  Zero is interpreted to mean 16.  There are 16 HW blocks.
    The block size is 98304 bytes.

    Each loaded database must have at least one HW block allocated to it.

    The initial load must be for database identifier 0.

    The allocation of HW blocks between database identifiers 0 and 1 is
    completely and permanently determined by the initial load of identifier 0:
    If the initial load allocates N HW blocks to identifier 0, then 16-N HW
    blocks are permanently allocated to identifier 1 (where 16 is the total
    number of HW blocks).  If all HW blocks are allocated to identifier 0,
    then none remain for allocation to identifier 1.  Hence, identifier 1 is
    effectively disabled and the resulting system has only a single database.
    This is not an error:  The application has the option of configuring
    itself as either a single or a dual database system.

Device limits

    1. At a given time, a flow can be bound with at most one database-id, group-id
    pair. For stateless flows, the pair can be changed for different
    searches but for stateful flows it cannot be changed.

NLS105/NLS205 DEVICES (mars2)
---------------------

Configuration parameters (struct nlm_device_config members)

    The configuration model of the NLS105/NLS205 devices is identical to the
    configuration model of the NLS025/NLS055 devices.

Device parameters (nlm_device_get/set_param arguments)

    The size of one database block is 8192 entities, where the size of an entry
```

```
    is 12 bytes.  (One database block is 8192 * 12 == 98304 bytes.)  There are
    exactly 16 HW database blocks.

                                         get         set
    NLM_FLOW_OFFSET                       y           y
    NLM_HW_REGISTER                       y           y
    NLM_MAX_MATCHES_PER_JOB               y           y
    NLM_MAX_STATES_PER_BYTE               y           y
    NLM_OUTPUT_FIFO_TIMER                 y           y
    NLM_INPUT_FIFO_TIMER                  y           y
    NLM_TIMESTAMP                         y           -
    NLM_DATABASE_LOAD_POLICY              y           y
    NLM_DATABASE_BLOCK_CNT                y           -
    NLM_DATABASE_TOTAL_BLOCK_CNT          y           -
    NLM_TOTAL_BLOCK_CNT                   y           -
    NLM_DATABASE_COPY_CNT                 -           -
    NLM_DATABASE_LOADER_STATUS            -           -
    NLM_PACKET_ENQUEUE_POLICY             -           -

Database load model.

    nlm_device_try_load_database (device, database, size, replacing_id, target_id,
                                  num_blocks)
    nlm_device_load_database (device, database, size, target_id, num_blocks)
    nlm_device_unload_database (device, target_id)

    There are exactly eight database identifiers (0 through 7).

    replacing_id can be any valid database identifier, or -1 (no identifier).

    num_blocks specifies a number of HW blocks.  The interpretation of
    num_blocks varies with the load policy (NLM_DATABASE_LOAD_POLICY).
    There are 16 HW blocks.  The block size is 98304 bytes.

    There is no requirement that a loaded database have at least one HW block
    allocated to it.

    There is no constraint on the database loading order relative to database
    identifier.

    Both stop scan and nonstop scan usage models are supported. NLS105/NLS205
    devices support a nonstop scan model much better than do NLS025/NLS055 devices,
    since the NLS105/NLS205 devices have a greater number of database identifiers and
    the distribution of HW blocks across the loaded databases can be varied
    arbitrarily with having to restart the DP.

Device limits

    The device limits of the NLS105/NLS205 devices are identical to the
    device limits of the NLS025/NLS055 devices.
```

```
NLS2008 (famos)
-------

Device parameters (nlm_device_get/set_param arguments)

    The size of one database block is 16384 entities, where each entity
    roughly corresponds to matching 1-byte in a rule.  There are
    exactly 12 HW database blocks.

                                      get         set
    NLM_FLOW_OFFSET                    y           y
    NLM_HW_REGISTER                    y           y
    NLM_MAX_MATCHES_PER_JOB            y           y
    NLM_MAX_STATES_PER_BYTE            -           -
    NLM_OUTPUT_FIFO_TIMER              y           y
    NLM_INPUT_FIFO_TIMER               y           y
    NLM_TIMESTAMP                      y           -
    NLM_DATABASE_LOAD_POLICY           y           y
    NLM_DATABASE_BLOCK_CNT             -           -
    NLM_DATABASE_TOTAL_BLOCK_CNT       -           -
    NLM_TOTAL_BLOCK_CNT                y           -
    NLM_DATABASE_COPY_CNT              y           y
    NLM_DATABASE_LOADER_STATUS         y           -
    NLM_PACKET_ENQUEUE_POLICY          y           y

Database load model.

    nlm_device_try_load_database (device, database, size, replacing_id, target_id,
                                  num_blocks)
    nlm_device_load_database (device, database, size, target_id, num_copies)
    nlm_device_unload_database (device, target_id)

    The load model of NLS2008 is different from that of NLS025/NLS105 devices
    because of underlying architectural differences.

    The nlm_device_try_load_database API only checks the validity of the input
    database and does not initiate any action to commence loading it to
    the device.

    The interpretation of the first four arguments of nlm_device_load_database
    is identical to NLS025/NLS205. The last argument is specifically related
    to optimizing throughput for this device. The NLS2008 has the
    property of maximizing throughput when 4 instances of a database exist
    in the device memory, with 3, 2, 1 instances leading to proportional
    decline in performance. This leads to the notion of managing the device
    memory with respect to the databases loaded on it.
    In general, the application has the knowledge of the search intensity
    for different databases and it can use this information to
    instantiate suitable number of copies of different databases. The last
    argument of nlm_device_load_database is for this purpose - to indicate
    the number of instances of a database. If num_copies = 1, 2, 3, 4, the
```

```
    loader will instantiate as many copies of the database in the device
    memory, subject to availability of space. Additionally, the loader
    supports a feature where it tries to maximize the number of instances
    of the database based on available memory. This is enabled by passing
    in 0 (zero) as the value of num_copies.

    If the available device database memory is not sufficient
    to load a new database, the application can choose to shrink the number
    of instances of an already loaded database. This can create some space
    for the new database at the expense of throughput of flows searching
    with the database that was shrunk.
    nlm_device_set_param (NLM_DATABASE_COPY_CNT, ...) can be used
    to shrink the number of copies an already loaded database.

Rings

    NLS2008 has a concept called a ring. Each ring is approximately an
    independent interface to the device. There are 4 PCIe rings (ring 0, 1, 2
    and 3) and 2 XAUI rings (ring 4 and 5). Ring 0 is special in that it is
    used to manage the rule databases, ring 0 is called the master ring. Ring 0
    must be the first ring to be opened.

    All rings must have a separate input fifo, output fifo, and context save /
    restore area. The memory requirements for the different regions of memory
    (contiguous physical address for sysmem, etc.) are the same. The DP
    enforces the requirement that the memory regions for each ring must not
    overlap with any other ring. Each XAUI ring also has a separate input and
    output fifo area but they are internal to the chip and not accessible
    through the DP interface. The results from a search job will be returned to
    the same ring that was used to enqueue the job. That is if a job is
    enqueued in ring 2 all its results will be returned to ring 2. The DP does
    not support any mixing of jobs and results from different rings.

    The chip does dynamic load balancing to allocate resources fairly to all
    the rings.

    Rings were introduced to allow independent applications to simultaneously
    access the device. For example one ring might be used for intrusion
    detection scanning and another could be used for anti-virus scanning.

    The master ring is accessed via the normal DP nlm_device_init() call. A new
    field in the config struct has been added to support the multi-ring
    interface: ring_id.  For the master ring this should be set to
    NLM_MASTER_RING.

    Any non-master ring can be accessed via the new DP function:
    nlm_device_attach().  For this call ring_id must have a ring_id other than
    NLM_MASTER_RING.  Use the new DP function nlm_device_detach() to end the
    use of a ring.

    Upto a maximum of 256 threads are supported on each ring. When attaching
```

```
    for the first time for ring_id other than NLM_MASTER_RING the
    thread_id must be zero. All other DP threads can then attach to the
    ring. The conf structure specified by all DP threads on a specific ring
    must be identical except for the thread_id. Similarly detach of
    thread_id = 0 on other than NLM_MASTER_RING must be the last call
    after all other DP threads have detached.

    In general there is no requirement that all the rings be accessed from
    within the same process. Nor is there a requirement that each ring
    be accessed from a different process.

Device limits

    1. At a given time, a flow can be bound with at most six database-id, group-id
    pairs. This family of devices permits arbitrary additions/removals
    of pairs, as long as the limit of 6 concurrent pairs is not exceeded.


Calling API's from different DP threads
---------------------------------------

Not all DP API functions can be called by all threads on all rings, the
following matrix enumerates when the API's can be used.


                                 thread_id = 0  thread_id = 0   thread_id = 1..255  Mana
                                  ring_id = 0    ring_id != 0    Any ring_id        any
nlm_device_init                     Y              -               -                -
nlm_device_fini                     Y              -               -                -

nlm_device_attach                   -              Y               Y                Y
nlm_device_detach                   -              Y               Y                Y

nlm_device_try_load_database        Y              -               -                -
nlm_device_load_database            Y              -               -                -
nlm_device_get_param                Y              Y               -                -
nlm_device_set_param                Y              Y               -                -

nlm_create_flow                     Y              Y               Y                -
nlm_flow_add_database_and_group     Y              Y               Y                -
nlm_flow_remove_database_and_group  Y              Y               Y                -
nlm_destroy_stateful_flow           Y              Y               Y                -
nlm_destroy_stateless_flow          Y              Y               Y                -
nlm_flow_enqueue_search             Y              Y               Y                -
nlm_cancel_job                      Y              Y               Y                -
nlm_start_jobs                      Y              Y               Y                -
nlm_get_all_search_results          Y              Y               Y                -

nlm_manage_input                    -              -               -                Y
nlm_manage_output                   -              -               -                Y
```

```
Typical API Usage
-----------------

Main Thread 0:
--------------

   {
     struct nlm_device_config master_config;
     struct nlm_device *master_device;

     nlm_estimate_device_config (n_flows, n_jobs, &master_config)
     master_config.max_threads = 2 /* We want to use 2 DP threads */
     master_config.thread_id = 0 /* Only thread_id = 0 can call device init */
     master_config.using_manager_thread = 1; /* Plan to use manager thread optional */
     /* Set other config parametres */

     nlm_device_init (&master_config, &master_device);
     nlm_device_load_database (master_device, database, size, 0, 0);

     /* Can set/reset device parameters, reload database
        using the master_device handle */

     while (...)
       {
         nlm_flow_enqueue_search (master_device, ...);
         nlm_start_jobs (master_device, ...);
         nlm_get_all_search_results (master_device, ...);
       }

     /* Wait for all other DP threads to complete */

     nlm_device_fini (master_device);
   }

Remaining DP threads:
---------------------
   {
     struct nlm_device *slave;
     /* Configuration is a copy of the master */
     struct nlm_device_config config = master_config;
     config.thread_id = 1; /* Place relevant thread_id */

     /* Call attach to initialize ourselves */
     nlm_device_attach (&config, &slave);

     while (...)
       {
         nlm_flow_enqueue_search (slave, ...);
         nlm_start_jobs (slave, ...);
         nlm_get_all_search_results (slave, ...);
       }
```

```
    /* Detach to free up resources */
    nlm_device_detach (slave);
  }

Manager thread (single thread configuration):
---------------------------------------------
  {
    struct nlm_device *manager;
    /* Configuration is a copy of the master */
    struct nlm_device_config config = master_config;
    config.thread_id = NLM_MANAGER_THREAD_ID;

    /* Call attach to initialize ourselves */
    nlm_device_attach (&config, &manager);

    while (...)
      {
        nlm_manage_input (manager);
        nlm_manage_output (manager);
      }

    /* Detach to free up resources */
    nlm_device_detach (manager);
  }

Manager thread (two thread configuration):
---------------------------------------------

Manager Thread 1:

  {
    struct nlm_device *manager;
    /* Configuration is a copy of the master */
    struct nlm_device_config config = master_config;
    config.thread_id = NLM_MANAGER_THREAD_ID;

    /* Call attach to initialize ourselves */
    nlm_device_attach (&config, &manager);

    while (...)
      {
        nlm_manage_input (manager);
      }

    /* Detach to free up resources */
    nlm_device_detach (manager);
  }

Manager Thread 2:
```

```
{
  struct nlm_device *manager;
  /* Configuration is a copy of the master */
  struct nlm_device_config config = master_config;
  config.thread_id = NLM_MANAGER_THREAD_ID;

  /* Call attach to initialize ourselves */
  nlm_device_attach (&config, &manager);

  while (...)
    {
      nlm_manage_output (manager);
    }

  /* Detach to free up resources */
  nlm_device_detach (manager);
}
```

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 nlm_device_config Struct Reference

structure to pass configuration parameters to data plane library

`#include <nlm_packet_api.h>`

Data Fields

- void * register_map_base_virt

  virt base addr of register map

- nlm_phys_addr register_map_base_phys

  phys base addr of register map

- uint32_t register_map_size

  size of register map in bytes

- void * sysmem_base_virt

  virt base addr of system memory

- nlm_phys_addr sysmem_base_phys

  phys base addr of system memory

- uint32_t sysmem_size

  size of system memory in bytes

- uint32_t input_fifo_size

  num of elements in input fifo

- uint32_t output_fifo_size

  num of elements in output fifo

- uint32_t size_of_database_extension_area

  database spill over size in bytes.

- uint32_t size_of_context_save_restore_area

  context save/restore area in bytes

- nlm_phys_addr(∗ virt_to_phys)(const void ∗)

  convert virtual address to physical

- nlm_phys_addr packet_base_phys

  phys base addr of packet storage area

- const void ∗ packet_base_virt

  virt base addr of packet storage area

- void ∗ memory_pool

  user supplied memory pool

- uint32_t memory_pool_size

  size of memory pool in bytes

- int(∗ try_yield_cpu)(void)

  function to try to yield cpu

- nlm_phys_addr base_addr_of_ddr_memory

  NLS2008 specific configuration parameters.

- uint64_t size_of_ddr_memory

  size of attached DDR memory if non zero, it should be bigger than size_of_context_save_-restore_area

- nlm_ring_id ring_id

0 - master ring 1,2,3 - slave pcie rings 4 - xaui0 5 - xaui1 (For mars take values in nlm_vring_id

- void * cookie

  handle to be passed to xmalloc, xfree functions.

- void *(* xmalloc)(void *cookie, uint32_t)

  callback function to allocate virtual memory.

- void(* xfree)(void *cookie, void *ptr)

  callback function to release virtual memory allocated by call to xmalloc above.

- uint32_t max_threads

  Maximum number of dataplane threads.

- uint32_t thread_id

  Thread ID to which this config belongs.

- uint32_t using_manager_thread

  Will use upto 2 threads for managing the device fifo's.

- int32_t reserved1

  Reserved for Device driver/Dataplane communication.

4.1.1 Detailed Description

structure to pass configuration parameters to data plane library

4.1.2 Field Documentation

4.1.2.1 uint32_t nlm_device_config::size_of_database_extension_area

database spill over size in bytes.

Ignored by NLS2008

4.1.2.2 nlm_phys_addr nlm_device_config::base_addr_of_ddr_memory

NLS2008 specific configuration parameters.

base address of DDR memory (up to 64Gbyte)

4.1.2.3 void∗ nlm_device_config::cookie

handle to be passed to xmalloc, xfree functions.

Not interpreted by datapalne

4.1.2.4 void∗(∗ nlm_device_config::xmalloc)(void ∗cookie, uint32_t)

callback function to allocate virtual memory.

If specified, memory_pool pointer should be NULL. For each thread at most memory_pool_size of memory will be allocated using this callback. It is assumed this callback function is MT safe

4.1.2.5 void(∗ nlm_device_config::xfree)(void ∗cookie, void ∗ptr)

callback function to release virtual memory allocated by call to xmalloc above.

It is assumed this function is MT safe.

4.1.2.6 uint32_t nlm_device_config::max_threads

Maximum number of dataplane threads.

Default 1

The documentation for this struct was generated from the following file:

- nlm_packet_api.h

4.2 nlm_result Struct Reference

Information about one result found by HW filled in by nlm_get_all_search_results() function.

`#include <nlm_packet_api.h>`

Data Fields

- nlm_status status

  status of this result.

- void ∗ cookie

  cookie for this result

- uint32_t rule_id

  rule ID that triggered the match

- uint32_t group_id

  group ID that triggered the match

- uint32_t database_id

  database ID that triggered the match

- uint32_t byte_offset

  match offset in bytes from the beginning of the job pointing to the last byte of the match

- uint64_t flow_offset

  match offset in bytes from the beginning of the flow pointing to the last byte of the match

- uint32_t match_length

  length of the match in bytes

- uint64_t timestamp

  HW timestamp when end_of_job result goes into ouput fifo.

- uint32_t total_state_cnt

  total number of states executed by HW engine

- uint8_t peak_state_cnt

peak number of states seen by HW engine

- uint8_t final_state_cnt

  final number of states seen by HW engine

4.2.1 Detailed Description

Information about one result found by HW filled in by nlm_get_all_search_results() function.

4.2.2 Field Documentation

4.2.2.1 nlm_status nlm_result::status

status of this result.

- `NLM_OK` u.match.* fields contain info about match
- `NLM_END_ANCHORED` u.match.* fields contain info about match that needs to be post-processed, since it was triggered by end-anchored rule
- `NLM_END_OF_JOB` u.stats.* fields contain statistics for completed job
- `NLM_*` u.* is undefined and status contains the error code for abnormally completed job

The documentation for this struct was generated from the following file:

- nlm_packet_api.h

Chapter 5

File Documentation

5.1 nlm_common_api.h File Reference

header file for shared structures in Dataplane/packet API and Controlplane/database API

```
#include "nlm_stdint.h"
#include "nlm_error_tbl.def"
```

Typedefs

- typedef unsigned long long nlm_phys_addr

  defines physical address for data plane library

Enumerations

- enum nlm_status

  status and error codes returned by all API functions

Functions

- const char $*$ nlm_get_status_string (nlm_status status)

  function to convert status code to string

5.1.1 Detailed Description

header file for shared structures in Dataplane/packet API and Controlplane/database API

5.1.2 Function Documentation

5.1.2.1 const char* nlm_get_status_string (nlm_status *status*)

function to convert status code to string

Parameters:

← ***status*** code to be converted

Returns:

string that describes the code

5.2 nlm_driver_api.h File Reference

header file for linux specific initialization phase of Data plane API

`#include "nlm_packet_api.h"`

Functions

- nlm_status nlm_get_device_count (uint32_t *n_devices)

  Get number of netl7 devices.

- nlm_status nlm_prepare_device_config (uint32_t device_id, struct nlm_device_config *config)

  Initialize netl7 device configuration Fill in platform specific fields of the configuration like system memory, register map, packet memory addresses.

- nlm_status nlm_free_device_config (struct nlm_device_config *config)

  Unmap virtual memory mapped by nlm_prepare_device_config() for system memory, packet memory and register memory.

- nlm_status nlm_init_all_devices (struct nlm_device *devices[NLM_MAX_DEVICE_COUNT], uint32_t *n_devices)

  Initialize all netl7 devices.

- nlm_status nlm_fini_all_devices (struct nlm_device *devices[NLM_MAX_DEVICE_COUNT], uint32_t n_devices)

  Shutdown all netl7 devices.

5.2.1 Detailed Description

header file for linux specific initialization phase of Data plane API

5.2.2 Function Documentation

5.2.2.1 nlm_status nlm_get_device_count (uint32_t * *n_devices*)

Get number of netl7 devices.

Parameters:

→ ***n_devices*** number of the netl7 devices on the system

Returns:

status

5.2.2.2 nlm_status nlm_prepare_device_config (uint32_t *device_id*, struct nlm_device_config ∗ *config*)

Initialize netl7 device configuration Fill in platform specific fields of the configuration like system memory, register map, packet memory addresses.

Parameters:

← ***device_id*** id of the netl7 device on the system

← ***config*** device configuration

→ ***config*** device configuration

Returns:

status

5.2.2.3 nlm_status nlm_free_device_config (struct nlm_device_config ∗ *config*)

Unmap virtual memory mapped by nlm_prepare_device_config() for system memory, packet memory and register memory.

Free memory pool

5.2.2.4 nlm_status nlm_init_all_devices (struct nlm_device ∗ *devices*[NLM_MAX_-DEVICE_COUNT], uint32_t ∗ *n_devices*)

Initialize all netl7 devices.

Parameters:

→ ***devices*** array of initialized device handles

→ ***n_devices*** number of initialized devices

Returns:

status

5.2.2.5 nlm_status nlm_fini_all_devices (struct nlm_device $*$ *devices*[NLM_MAX_-DEVICE_COUNT], uint32_t *n_devices*)

Shutdown all netl7 devices.

Parameters:

← ***devices*** array of device handles to be shutdown

← ***n_devices*** number of devices

Returns:

status

5.3 nlm_packet_api.h File Reference

header file for Dataplane/packet API

```
#include "nlm_common_api.h"
```

Data Structures

- struct nlm_device_config

  structure to pass configuration parameters to data plane library

- struct nlm_result

  Information about one result found by HW filled in by nlm_get_all_search_results() function.

Defines

- #define DEFAULT_NLM_DEVICE_CONFIG

  macro with default initialization values

Enumerations

- enum nlm_device_param {

  NLM_INVALID_PARAM = NLM_FIRST_DEVICE_PARAM, NLM_FLOW_OFFSET, NLM_HW_MEMORY, NLM_HW_REGISTER,

  NLM_MAX_MATCHES_PER_JOB, NLM_MAX_STATES_PER_BYTE, NLM_-OUTPUT_FIFO_TIMER, NLM_INPUT_FIFO_TIMER,

  NLM_TIMESTAMP, NLM_DATABASE_COPY_CNT, NLM_DATABASE_BLOCK_-CNT, NLM_DATABASE_TOTAL_BLOCK_CNT,

  NLM_DATABASE_BALANCE, NLM_DATABASE_LOAD_POLICY, NLM_-DATABASE_LOADER_STATUS, NLM_TOTAL_BLOCK_CNT,

  NLM_PACKET_ENQUEUE_POLICY, NLM_FLOW_FORCE_ONE_FD }

  configuration parameters that can be set and get by corresponding functions

- enum nlm_flow_type { NLM_FLOW_TYPE_INVALID = NLM_FIRST_FLOW_TYPE, NLM_FLOW_TYPE_STATEFUL, NLM_FLOW_TYPE_STATELESS }

  supported flow types

- enum nlm_database_load_policy { NLM_DATABASE_LOAD_POLICY_DEFAULT = 0, NLM_DATABASE_LOAD_POLICY_MANUAL, NLM_DATABASE_LOAD_-POLICY_ASYNC }

  supported database load policies

Functions

- nlm_status nlm_estimate_device_config (uint32_t n_flows, uint32_t n_jobs, struct nlm_-device_config ∗config)

  Estimate netl7 device configuration parameters based on number of flows and jobs and fill in different ∗_size fields of configuration.

- nlm_status nlm_device_init (struct nlm_device_config ∗config, struct nlm_device ∗∗p_-device)

  Initialize device with 'config' and return pointer to it.

- nlm_status nlm_device_fini (struct nlm_device ∗device)

  Shutdown device.

- nlm_status nlm_device_attach (struct nlm_device_config ∗config, struct nlm_device ∗∗p_-device)

  Attach to initialized device with 'config' and return pointer to it.

- nlm_status nlm_device_detach (struct nlm_device ∗device)

  Detach from the live device without shutting it down.

- nlm_status nlm_device_try_load_database (struct nlm_device ∗device, const void ∗database, uint32_t db_size, uint32_t replacing_db_id, uint32_t new_db_id, uint32_-t num_blocks)

  Try loading the database.

- nlm_status nlm_device_load_database (struct nlm_device ∗device, const void ∗database, uint32_t db_size, uint32_t database_id, uint32_t num_blocks)

  Load database onto the device.

- nlm_status nlm_device_unload_database (struct nlm_device ∗device, uint32_t database_-id)

Unload database and free associated memory.

- nlm_status nlm_device_get_param (struct nlm_device *device, nlm_device_param param,...)

  Get specified device parameter.

- nlm_status nlm_device_set_param (struct nlm_device *device, nlm_device_param param,...)

  Set specified device parameter.

- nlm_status nlm_create_flow (struct nlm_device *device, nlm_flow_type flow_type, struct nlm_flow **p_flow)

  Create flow of the type 'flow_type' on the 'device' for subsequent search in 'database_id'.

- nlm_status nlm_flow_add_database_and_group (struct nlm_device *device, struct nlm_-flow *flow, uint32_t database_id, uint32_t group_id)

  add (database, rule_group) pair to the set of (database, group) pairs searched in the following packets of this flow.

- nlm_status nlm_flow_remove_database_and_group (struct nlm_device *device, struct nlm_flow *flow, uint32_t database_id, uint32_t group_id)

  remove (database, rule_group) pair to the set of (database, group) pairs searched in the following packets of this flow.

- nlm_status nlm_destroy_stateful_flow (struct nlm_device *device, struct nlm_flow *flow, void *cookie)

  Destroy 'flow' on the 'device'.

- nlm_status nlm_destroy_stateless_flow (struct nlm_device *device, struct nlm_flow *flow)

  Destroy 'flow' on the 'device'.

- nlm_status nlm_flow_enqueue_search (struct nlm_device *device, struct nlm_flow *flow, const void *start, const void *end, void *cookie, struct nlm_job **p_job)

  Enqueue payload for the search from 'start' to 'end' in 'flow' with 'group_id' and 'cookie' which is going to be accepted as-is and returned by nlm_get_all_search_results() Function returns 'job' pointer for this search request.

- nlm_status nlm_cancel_job (struct nlm_device *device, struct nlm_job *job)

  Cancel 'job' that wasn't sent to the 'device'.

- nlm_status nlm_start_jobs (struct nlm_device ∗device, uint32_t n_jobs, struct nlm_job ∗jobs[ ])

  Start searching 'jobs' on the 'device'.

- nlm_status nlm_get_all_search_results (struct nlm_device ∗device, uint32_t n_buf_-entries, struct nlm_result ∗buf, uint32_t ∗n_results)

  Return all results found so far by device *into* buf *buffer, but no more than* n_buf_entries *at a time.*

- nlm_status nlm_manage_input (struct nlm_device ∗device)

  Executed by the manager thread.

- nlm_status nlm_manage_output (struct nlm_device ∗device)

  Executed by the manager thread.

5.3.1 Detailed Description

header file for Dataplane/packet API

5.3.2 Enumeration Type Documentation

5.3.2.1 enum nlm_device_param

configuration parameters that can be set and get by corresponding functions

Enumerator:

NLM_INVALID_PARAM marker for invalid parameter

NLM_FLOW_OFFSET offset of the last byte processed in the flow

NLM_HW_MEMORY HW memory.

NLM_HW_REGISTER HW register.

NLM_MAX_MATCHES_PER_JOB maximum number of matches per job

NLM_MAX_STATES_PER_BYTE maximum number of states per byte

NLM_OUTPUT_FIFO_TIMER HW timer controls when the internal result fifo gets flushed out to system memory.

NLM_INPUT_FIFO_TIMER HW timer that must pass before the DMA engine updates the external input fifo read and output fifo write pointers in system memory.

NLM_TIMESTAMP HW timestamp counter.

NLM_DATABASE_COPY_CNT number of database copies in the device

NLM_DATABASE_BLOCK_CNT number of blocks loaded into HW memory for given database

NLM_DATABASE_TOTAL_BLOCK_CNT total number of blocks for given database total == number of blocks in HW memory + blocks in spillover

NLM_DATABASE_BALANCE Balance loaded databases, set only parameter.

NLM_DATABASE_LOAD_POLICY database load/unload policy

NLM_DATABASE_LOADER_STATUS info on the current loader state

NLM_TOTAL_BLOCK_CNT total number of database blocks supported by HW

NLM_PACKET_ENQUEUE_POLICY packet enqueue policy

NLM_FLOW_FORCE_ONE_FD force FD to be generated for the next start_jobs()

5.3.2.2 enum nlm_flow_type

supported flow types

Enumerator:

NLM_FLOW_TYPE_INVALID marker for invalid flow

NLM_FLOW_TYPE_STATEFUL stateful flow

NLM_FLOW_TYPE_STATELESS stateless flow

5.3.2.3 enum nlm_database_load_policy

supported database load policies

Enumerator:

NLM_DATABASE_LOAD_POLICY_DEFAULT default load policy for this device

NLM_DATABASE_LOAD_POLICY_MANUAL user specified load

NLM_DATABASE_LOAD_POLICY_ASYNC FMS asynchronous load model.

5.3.3 Function Documentation

5.3.3.1 nlm_status nlm_estimate_device_config (uint32_t *n_flows*, uint32_t *n_jobs*, struct nlm_device_config ∗ *config*)

Estimate netl7 device configuration parameters based on number of flows and jobs and fill in different ∗_size fields of configuration.

Parameters:

← ***n_flows*** number of flows

← ***n_jobs*** number of jobs

→ ***config*** device configuration

Returns:

status

5.3.3.2 nlm_status nlm_device_init (struct nlm_device_config ∗ *config*, struct nlm_device ∗∗ *p_device*)

Initialize device with 'config' and return pointer to it.

Parameters:

← ***config*** device configuration parameters as specified in nlm_device_config structure

→ ***p_device*** returned device handle. This pointer should not be zero

Returns:

status

Warning:

Should be called only by thread_id = 0

5.3.3.3 nlm_status nlm_device_fini (struct nlm_device ∗ *device*)

Shutdown device.

Parameters:

← ***device*** device handle

Returns:

status

Warning:

should be called only by thread_id = 0 after all remaining DP threads have called nlm_-device_detach

5.3.3.4 nlm_status nlm_device_attach (struct nlm_device_config * *config*, struct nlm_device ** *p_device*)

Attach to initialized device with 'config' and return pointer to it.

Parameters:

← ***config*** device configuration parameters as specified in nlm_device_config structure

→ ***p_device*** returned device handle. This pointer should not be zero

Returns:

status

5.3.3.5 nlm_status nlm_device_detach (struct nlm_device * *device*)

Detach from the live device without shutting it down.

Parameters:

← ***device*** device handle

Returns:

status

5.3.3.6 nlm_status nlm_device_try_load_database (struct nlm_device * *device*, const void * *database*, uint32_t *db_size*, uint32_t *replacing_db_id*, uint32_t *new_db_id*, uint32_t *num_blocks*)

Try loading the database.

Parameters:

← ***device*** device handle

← ***database*** pointer to compiled rule image

← ***db_size*** database buffer size

← ***replacing_db_id*** database id the new database is about to replace

← ***new_db_id*** database id of the new database

← ***num_blocks*** number of physical HW blocks to be occupied by database

Returns:

NLM_OK if the database can be successfully loaded or error code otherwise

Warning:

should be called only by thread_id = 0

5.3.3.7 nlm_status nlm_device_load_database (struct nlm_device ∗ *device*, const void ∗ *database*, uint32_t *db_size*, uint32_t *database_id*, uint32_t *num_blocks*)

Load database onto the device.

this function will call yield_cpu() callback from time to time to yield cpu to accommodate OSes with cooperative multi-tasking

Parameters:

← ***device*** device handle

← ***database*** pointer to compiled rule image

← ***db_size*** database buffer size

← ***database_id*** id to be assigned to this database that further will be used in nlm_device_-unload_database() and nlm_create_flow() functions

← ***num_blocks*** number of physical HW blocks to be occupied by database

Returns:

NLM_OK if database is fully loaded and error code otherwise

Warning:

should be called only by thread_id = 0

5.3.3.8 nlm_status nlm_device_unload_database (struct nlm_device * *device*, uint32_t *database_id*)

Unload database and free associated memory.

Parameters:

← ***device*** device handle

← ***database_id*** id of the database to be unloaded

Returns:

status

Warning:

should be called only by thread_id = 0

5.3.3.9 nlm_status nlm_create_flow (struct nlm_device * *device*, nlm_flow_type *flow_type*, struct nlm_flow ** *p_flow*)

Create flow of the type 'flow_type' on the 'device' for subsequent search in 'database_id'.

Parameters:

← ***device*** device handle

← ***flow_type*** type of flow to be created

→ ***p_flow*** pointer to the created flow

Returns:

status

5.3.3.10 nlm_status nlm_flow_add_database_and_group (struct nlm_device * *device*, struct nlm_flow * *flow*, uint32_t *database_id*, uint32_t *group_id*)

add (database, rule_group) pair to the set of (database, group) pairs searched in the following packets of this flow.

Parameters:

← ***device*** device handle

← ***flow*** flow handle

← ***database_id*** database id to be used during the search

← ***group_id*** scan packet with this group id

Returns:

status

5.3.3.11 nlm_status nlm_flow_remove_database_and_group (struct nlm_device ∗ *device*, struct nlm_flow ∗ *flow*, uint32_t *database_id*, uint32_t *group_id*)

remove (database, rule_group) pair to the set of (database, group) pairs searched in the following packets of this flow.

Parameters:

← ***device*** device handle

← ***flow*** flow handle

← ***database_id*** database id used during the search

← ***group_id*** remove this group id

Returns:

status

5.3.3.12 nlm_status nlm_destroy_stateful_flow (struct nlm_device ∗ *device*, struct nlm_flow ∗ *flow*, void ∗ *cookie*)

Destroy 'flow' on the 'device'.

Sends visible destroy/finish flow request with given *cookie*

Parameters:

← ***device*** device handle

← ***flow*** flow handle

← ***cookie*** is going to be accepted as-is and returned by nlm_get_all_search_results()

Returns:

status

5.3.3.13 nlm_status nlm_destroy_stateless_flow (struct nlm_device ∗ *device*, struct nlm_flow ∗ *flow*)

Destroy 'flow' on the 'device'.

Sends hidden destroy/finish flow request to HW

Parameters:

← ***device*** device handle

← ***flow*** flow handle

Returns:

status

5.3.3.14 nlm_status nlm_flow_enqueue_search (struct nlm_device ∗ *device*, struct nlm_flow ∗ *flow*, const void ∗ *start*, const void ∗ *end*, void ∗ *cookie*, struct nlm_job ∗∗ *p_job*)

Enqueue payload for the search from 'start' to 'end' in 'flow' with 'group_id' and 'cookie' which is going to be accepted as-is and returned by nlm_get_all_search_results() Function returns 'job' pointer for this search request.

Parameters:

← ***device*** device handle

← ***flow*** flow handle

← ***start*** pointer to the first byte to be scanned

← ***end*** pointer to the byte after the last byte to be scanned

← ***cookie*** is going to be accepted as-is and returned by nlm_get_all_search_results()

→ ***p_job*** returns job pointer that tracks this search request. p_job pointer should not be zero

Returns:

status

5.3.3.15 nlm_status nlm_cancel_job (struct nlm_device * *device*, struct nlm_job * *job*)

Cancel 'job' that wasn't sent to the 'device'.

Parameters:

← ***device*** device handle

← ***job*** job handle to be cancelled

Returns:

status

5.3.3.16 nlm_status nlm_start_jobs (struct nlm_device * *device*, uint32_t *n_jobs*, struct nlm_job * *jobs*[ ])

Start searching 'jobs' on the 'device'.

Parameters:

← ***device*** device handle

← ***n_jobs*** number of jobs in the array

← ***jobs*** array of jobs to be started

Returns:

status

5.3.3.17 nlm_status nlm_get_all_search_results (struct nlm_device * *device*, uint32_t *n_buf_entries*, struct nlm_result * *buf*, uint32_t * *n_results*)

Return all results found so far by *device* into *buf* buffer, but no more than *n_buf_entries* at a time.

Parameters:

← ***device*** device handle

← ***n_buf_entries*** requested number of entries in the result buffer

← ***buf*** result buffer to store results

→ ***n_results*** actual number of results returned

Returns:

NLM_OK, if *n_results* were found and copied into *buf*, *n_results* can be zero, which means that no new results were found and *buf* is not changed

5.3.3.18 nlm_status nlm_manage_input (struct nlm_device * *device*)

Executed by the manager thread.

Dispatches accumulated jobs from DP threads to the physical device.

Parameters:

← ***device*** device handle

Returns:

NLM_OK, if no errors were encountered during submission of jobs to hardware

5.3.3.19 nlm_status nlm_manage_output (struct nlm_device * *device*)

Executed by the manager thread.

Return results produced by the physical device to the DP thread buffers for later retreival by the DP threads

Parameters:

← ***device*** device handle

Returns:

NLM_OK, if no errors were encountered during the the retreival of results from hardware

Index