fna Reference Manual

Generated by Doxygen 1.4.6

Wed Mar 7 09:56:59 2007

Contents

Chapter 1

fna Module Index

1.1 fna Modules

Here is a list of all modules:

Chapter 2

fna File Index

2.1 fna File List

Here is a list of all files with brief descriptions:

Chapter 3

fna Module Documentation

3.1 FNA Private Interface

Modules

- FNA Initialization
- FNA Virtual Resources
- FNA Spare Capacity
- FNA Send and Receive
- FNA Network Configuration

3.1.1 Detailed Description

FNA is a Network adaption layer that allows to plugin new network protocols to the distributed module.

It is divided in two parts:

- FRSH_FNA: public types and functions for the FRSH API
- FNA: private functions only used within FRSH.

3.2 FNA Initialization

Functions

- int fna_init (const frsh_resource_id_t resource_id)

3.2.1 Detailed Description

These functions need to be called before using any network

3.2.2 Function Documentation

3.2.2.1 int fna_init (const frsh_resource_id_t *resource_id*)

fna_init()

This function will be hooked to the frsh_init function and it is intented to initialize the protocol and its structures.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

Returns:

FNA_NO_ERROR
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_ALREADY_INITIALIZED: if the function has already been called before (with success)

3.3 FNA Virtual Resources

Typedefs

- typedef void * fna_vres_id_t

Functions

- int fna_vres_create (const frsh_resource_id_t resource_id, const frsh_contract_t *contract, fna_vres_id_t *vres)
- int fna_vres_renegotiate_sync (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, const frsh_contract_t *new_contract)
- int fna_vres_renegotiate_async (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, const frsh_contract_t *new_contract, frsh_signal_t signal_to_notify, frsh_signal_info_t signal_info)
- int fna_vres_get_renegotiation_status (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, frsh_renegotiation_status_t *renegotiation_status)
- int fna_vres_destroy (const frsh_resource_id_t resource_id, const fna_vres_id_t vres)
- int fna_vres_get_contract (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, frsh_contract_t *contract)
- int fna_vres_get_usage (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, struct timespec *usage)
- int fna_vres_get_remaining_budget (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, struct timespec *remaining_budget)
- int fna_vres_get_budget_and_period (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, struct timespec *budget, struct timespec *period)

3.3.1 Detailed Description

The following functions are used to negotiate, renegotiate and cancel virtual network resources.

3.3.2 Typedef Documentation

3.3.2.1 typedef void* fna_vres_id_t

fna_vres_id_t

Internal virtual resource id. The type fna_vres_id_t is a pointer to void. The FRSH layer will keep a map between the frsh_vres_id_t, this pointer and the resource_id. This pointer could be used as the ID itself using casting, or as internal pointer to any structure.

Definition at line 148 of file fna.h.

3.3.3 Function Documentation

3.3.3.1 int fna_vres_create (const frsh_resource_id_t *resource_id*, const frsh_contract_t * *contract*, fna_vres_id_t * *vres*)

fna_vres_create()

The operation negotiates a contract and if accepted it will return a fna_vres_id_t. It will also check that the given contract_id is unique within the network.

In object oriented terminology it is similar to a constructor of the virtual resource class.

If the on-line admission test is enabled, it determines whether the contract can be admitted or not based on the current contracts established in the network. Then it creates the vres and recalculates all necessary parameters for the contracts already present in the system.

This is a potentially blocking operation, it returns when the system has either rejected the contract, or admitted it and made it effective.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***contract*** The contract parameters to negotiate

→ ***vres*** The internal virtual resource id

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_TOO_MANY_VRES: if there is no space for more vres
FNA_ERR_CONTRACT_ID_ALREADY_EXISTS: contract_id is not unique
FNA_ERR_CONTRACT_REJECTED: if the contract is not accepted
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.3.3.2 int fna_vres_destroy (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*)

fna_vres_destroy()

The operation eliminates the specified vres and recalculates all necessary parameters for the contracts remaining in the system. This is a potentially blocking operation; it returns when the system has made the changes effective.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id to destroy

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.3.3.3 int fna_vres_get_budget_and_period (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*, struct timespec * *budget*, struct timespec * *period*)

fna_vres_get_budget_and_period()

This function gets the budget and period associated with the specified vres for each period. If one of these pointers is NULL, the corresponding information is not stored.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id

→ ***budget*** The budget associated to vres

→ ***period*** The period associated to vres

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if both pointers are NULL

3.3.3.4 int fna_vres_get_contract (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*, frsh_contract_t * *contract*)

fna_vres_get_contract()

This operation stores the contract parameters currently associated with the specified vres in the variable pointed to by contract. It returns an error if the vres_id is not recognised.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id

→ ***contract*** The contract parameters that we want

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.3.3.5 int fna_vres_get_remaining_budget (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*, struct timespec * *remaining_budget*)

fna_vres_get_remaining_budget()

This function stores in the variable pointed to by budget the remaining execution-time budget associated with the specified vres in the present period.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id

→ ***remaining_budget*** The remaining budget for this period

Returns:
FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.3.3.6 int fna_vres_get_renegotiation_status (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*, frsh_renegotiation_status_t * *renegotiation_status*)

fna_vres_get_renegotiation_status()

The operation reports on the status of the last renegotiation operation enqueued for the specified vres. It is callable even after notification of the completion of such operation, if requested.

If the vres is not and has not been involved in any of the frsh_contract_renegotiate_async() or frsh_group_-change_mode_async() operations, the status returned is FNA_NOT_REQUESTED

Parameters:
← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id we want the status from

← ***renegotiation_status*** The status of the last renegotiation on vres (FRSH_RS_IN_PROGRESS, FRSH_RS_REJECTED, FRSH_RS_ADMITTED, FRSH_RS_NOT_REQUESTED)

Returns:
FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.3.3.7 int fna_vres_get_usage (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*, struct timespec * *usage*)

fna_vres_get_usage()

This function gets the execution time spent by all messages that have been sent through the specified vres.

Parameters:
← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id

→ ***usage*** Execution time spent by this vres

Returns:
FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted

FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.3.3.8 int fna_vres_renegotiate_async (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*, const frsh_contract_t * *new_contract*, frsh_signal_t *signal_to_notify*, frsh_signal_info_t *signal_info*)

fna_vres_renegotiate_async()

The operation enqueues a renegotiate operation for an existing vres, and returns immediately. The renegotiate operation is performed asynchronously, as soon as it is practical; meanwhile the system operation will continue normally. When the renegotiation is made, if the on-line admission test is enabled it determines whether the contract can be admitted or not based on the current contracts established in the system. If it cannot be admitted, the old contract remains in effect. If it can be admitted, it recalculates all necessary parameters for the contracts already present in the system.

When the operation is completed, notification is made to the caller, if requested, via a signal. The status of the operation (in progress, admitted, rejected) can be checked with the frsh_vres_get_renegotiation_status() operation. The argument sig_notify can be FRSH_NULL_SIGNAL (no notification), or any FRSH signal value and in this case signal_info is to be sent with the signal.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id to renegotiate

← ***new_contract*** The new contract

← ***signal_to_notify*** Signal number to use to notify vres of the negotiation result. If FRSH_NULL_SIGNAL, no signal will be raised.

← ***signal_info,:*** Associated info that will come with the signal. This parameter will be ignored if signal_to_notify == FRSH_NULL_SIGNAL.

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted
FNA_ERR_CONTRACT_ID_ALREADY_EXISTS: contract_id is not unique
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL, or sig_notify is neither NULL nor a valid POSIX signal

3.3.3.9 int fna_vres_renegotiate_sync (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*, const frsh_contract_t * *new_contract*)

fna_vres_renegotiate_sync()

The operation renegotiates a contract for an existing vres. If the on-line admission test is enabled it determines whether the contract can be admitted or not based on the current contracts established in the system. If it cannot be admitted, the old contract remains in effect and an error is returned. If it can be admitted, it recalculates all necessary parameters for the contracts already present in the system and returns zero.

This is a potentially blocking operation; it returns when the system has either rejected the new contract, or admitted it and made it effective.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id to renegotiate

← ***new_contract*** The new contract

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted
FNA_ERR_CONTRACT_ID_ALREADY_EXISTS: contract_id is not unique
FNA_ERR_CONTRACT_REJECTED: if the contract is not accepted
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.4 FNA Spare Capacity

Functions

- int fna_spare_get_capacity (const frsh_resource_id_t resource_id, const int importance, uint32_t *capacity)
- int fna_spare_get_total_weight (const frsh_resource_id_t resource_id, const int importance, int *total_weight)
- int fna_spare_release_capacity (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, const struct timespec new_budget, const struct timespec new_period)

3.4.1 Detailed Description

The following functions are used to get spare capacity data

3.4.2 Function Documentation

3.4.2.1 int fna_spare_get_capacity (const frsh_resource_id_t *resource_id*, const int *importance*, uint32_t * *capacity*)

fna_spare_get_capacity()

This operation gets the spare capacity currently assigned to a importance level. If we divide this value by UINT32_MAX we will get the network utilization associated to the spare capacity of a importance level.

The following is typically in stdint.h:

- typedef unsigned int uint32_t;
- # define UINT32_MAX (4294967295U)

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***importance*** The importance we want the capacity of

→ ***capacity*** The spare capacity for that importance level

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.4.2.2 int fna_spare_get_total_weight (const frsh_resource_id_t *resource_id*, const int *importance*, int * *total_weight*)

fna_spare_get_total_weight()

This function gets the sum of the weight parameters for all vres in a network of an importance level.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***importance*** The importance we want the total weight of

→ ***total_weight*** The total weight for that importance level

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.4.2.3 int fna_spare_release_capacity (const frsh_resource_id_t *resource_id*, const fna_vres_id_t *vres*, const struct timespec *new_budget*, const struct timespec *new_period*)

fna_spare_release_capacity()

This function allows to ask for less budget and period than what we received. The request must be compatible with the rest of contract parameters of the vres. If we want to recover the released capacity we will need to renegotiate.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***vres*** The internal virtual resource id

← ***new_budget*** The new_budget

← ***new_period*** The new Period

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_CONTRACTED_VRES: if the vres is not contracted
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL
FNA_ERR_CONTRACT_REJECTED: if it is incompatible with the current contract

3.5 FNA Send and Receive

Functions

- int fna_send_sync (const frsh_send_endpoint_t *endpoint, const void *msg, const size_t size)
- int fna_send_async (const frsh_send_endpoint_t *endpoint, const void *msg, const size_t size)
- int fna_receive_sync (const frsh_receive_endpoint_t *endpoint, void *buffer, const size_t buffer_size, size_t *received_bytes)
- int fna_receive_async (const frsh_receive_endpoint_t *endpoint, void *buffer, const size_t buffer_size, size_t *received_bytes)
- int fna_send_endpoint_get_pending_messages (const frsh_send_endpoint_t *endpoint, int *number_of_pending_messages)
- int fna_receive_endpoint_get_pending_messages (const frsh_receive_endpoint_t *endpoint, int *number_of_pending_messages)

3.5.1 Detailed Description

The following functions are used to send and receive

3.5.2 Function Documentation

3.5.2.1 int fna_receive_async (const frsh_receive_endpoint_t * *endpoint*, void * *buffer*, const size_t *buffer_size*, size_t * *received_bytes*)

fna_receive_async()

This operation is similar to the previous one but it works in a non blocking (asynchronous) fashion. If no message is available it returns with error FNA_NO_MESSAGE.

Parameters:

← ***endpoint*** The receive endpoint we are receiving from. (resource_id is in the endpoint).

→ ***buffer*** Buffer for storing the received message

← ***buffer_size*** The size in bytes of this buffer

→ ***received_bytes*** The actual number of received bytes

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL
FNA_ERR_NO_SPACE: if the message size is bigger than the provided buffer.
FNA_NO_MESSAGE: if no messages are available in the queue.

3.5.2.2 int fna_receive_endpoint_get_pending_messages (const frsh_receive_endpoint_t * *endpoint*, int * *number_of_pending_messages*)

fna_receive_endpoint_get_pending_messages

This function tells the number of messages still pending in the receive endpoint queue

Parameters:

← ***endpoint*** The receive endpoint (resource_id is in the endpoint).

→ ***number_of_pending_messages*** The number of pending messages

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.5.2.3 int fna_receive_sync (const frsh_receive_endpoint_t * *endpoint*, void * *buffer*, const size_t *buffer_size*, size_t * *received_bytes*)

fna_receive_sync()

This operation is used to receive messages from the network with a blocking behavior (if there are no messages this operation blocks the calling thread).

When a message is available, it is copied to buffer (up to its size). The number of bytes copied is returned in received_bytes. The rest of the bytes of that message will be lost or not depending on the protocol (FNA_ERR_NO_SPACE will be returned if it is).

The function fails with FNA_ERR_NO_SPACE if the buffersize is too small for the message received. In this case the message is lost.

Messages arriving at a receiver buffer that is full will be handled according to the queueing policy of the endpoint (overwrite oldest, discard it,etc).

Parameters:

← ***endpoint*** The receive endpoint we are receiving from. (resource_id is in the endpoint).

→ ***buffer*** Buffer for storing the received message

← ***buffer_size*** The size in bytes of this buffer

→ ***received_bytes*** The actual number of received bytes

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL
FNA_ERR_NO_SPACE: if the message size is bigger than the provided buffer.

3.5.2.4 int fna_send_async (const frsh_send_endpoint_t * *endpoint*, const void * *msg*, const size_t *size*)

fna_send_async()

This operation sends a message stored in msg and of length size through the given send endpoint. The operation is non-blocking and returns immediately.

Parameters:

← ***endpoint*** The send endpoint we are sending through. It must be bound to a virtual resource (resource_id is in the endpoint).

← ***msg*** The message we want to send

← ***size*** The size in bytes of the message

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_BOUND: if endpoint is not bound to a valid vres
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL
FNA_ERR_TOO_LARGE: if the message is too large for the network protocol
FNA_ERR_BUFFER_FULL: if the message has been discarded because the queue is full (and does not have the policy FNA_QP_OLDEST)

3.5.2.5 int fna_send_endpoint_get_pending_messages (const frsh_send_endpoint_t * *endpoint*, int * *number_of_pending_messages*)

fna_send_endpoint_get_pending_messages()

This function tells the number of messages still pending

Parameters:

← ***endpoint*** The send endpoint (resource_id is in the endpoint).

→ ***number_of_pending_messages*** The number of pending messages

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.5.2.6 int fna_send_sync (const frsh_send_endpoint_t * *endpoint*, const void * *msg*, const size_t *size*)

fna_send_sync()

Similar to previous function but now the sending thread gets blocked until the message is already sent to the network.

Parameters:

← ***endpoint*** The send endpoint we are sending through. It must be bound to a virtual resource (resource_id is in the endpoint).

← ***msg*** The message we want to send

← ***size*** The size in bytes of the message

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_BOUND: if endpoint is not bound to a valid vres

FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL
FNA_ERR_TOO_LARGE: if the message is too large for the network protocol
FNA_ERR_BUFFER_FULL: if the message has been discarded because the queue is full (and does not have the policy FNA_QP_OLDEST)

3.6 FNA Network Configuration

Functions

- int fna_message_get_tx_time (const frsh_resource_id_t resource_id, const size_t nbytes, struct timespec *tx_time)
- int fna_message_get_max_size (const frsh_resource_id_t *resource_id, size_t *max_size)

3.6.1 Detailed Description

These functions are needed to set/get some network dependent values

3.6.2 Function Documentation

3.6.2.1 int fna_message_get_max_size (const frsh_resource_id_t * *resource_id*, size_t * *max_size*)

fna_message_get_max_size()

This operation gives the maximum number of bytes that can be sent in a single message through the network designated by resource_id. If the application needs to send bigger messages it will have to split them. Some protocols like TCP/IP are capable of sending large messages (and use fragmentation internally) but other protocols doesn't provide fragmentation features so the maximum size will be the MTU itself.

It is be used by the application to calculate the minimum and maximum budgets used in the preparation of network contracts.

Parameters:

← ***resource_id*** The network we want the tx time from.

→ ***max_size*** The maximum number of bytes for each message

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL or nbytes is bigger than the maximum message size

3.6.2.2 int fna_message_get_tx_time (const frsh_resource_id_t *resource_id*, const size_t *nbytes*, struct timespec * *tx_time*)

fna_message_get_tx_time()

This operation gives the worst case transmission time that it takes to send a message of the nbytes size through the network designated by resource_id, including any network overheads (fragmentation in packets, headers, retransmissions,...).

It is be used by the application to calculate the minimum and maximum budgets used in the preparation of network contracts.

Parameters:

← ***resource_id*** The network we want the tx time from.

← ***nbytes*** Number of bytes of the message

→ ***tx_time*** The transmission time for nbytes (including overheads)

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL or nbytes is bigger than the maximum message size

3.7 FNA Public Interface

Modules

- FNA Mapping Functions
- FNA Negotiation Service Parameters

3.7.1 Detailed Description

FNA is a Network adaption layer that allows to plugin new network protocols to the distributed module.

It is divided in two parts:

- FRSH_FNA: public types and functions for the FRSH API
- FNA: private functions only used within FRSH.

3.8 FNA Mapping Functions

Functions

- int frsh_rtep_map_network_address (const frsh_resource_id_t resource_id, const rtep_network_address_t *in_address, frsh_network_address_t *out_address)
- int frsh_rtep_map_stream_id (const frsh_resource_id_t resource_id, const rtep_channel_t *in_stream, frsh_stream_id_t *out_stream)

3.8.1 Detailed Description

These functions are needed to map network specific types to FRSH types. Instead of providing this mapping functions a static hardwired configuration could be used.

3.8.2 Function Documentation

3.8.2.1 int frsh_rtep_map_network_address (const frsh_resource_id_t *resource_id*, const rtep_network_address_t * *in_address*, frsh_network_address_t * *out_address*)

frsh_XXXX_map_network_address()

To map a XXXX protocol network address into a FRSH address. The protocol must keep this mapping consistent. Instead of using a function a hardwired mapping could be used.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***in_address*** The network address we want to map to a frsh address

→ ***out_address*** The FRSH abstract network address

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.8.2.2 int frsh_rtep_map_stream_id (const frsh_resource_id_t *resource_id*, const rtep_channel_t * *in_stream*, frsh_stream_id_t * *out_stream*)

frsh_XXXX_map_stream_id()

To map a XXXX protocol network stream, port, channel... into a FRSH stream. The protocol must keep this mapping consistent. Instead of using a function a hardwired mapping could be used.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***in_stream*** The network stream we want to map to a FRSH stream

→ ***out_stream*** The FRSH abstract network stream

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.9 FNA Negotiation Service Parameters

Functions

- int frsh_rtep_negotiation_messages_vres_renegotiate (const frsh_resource_id_t resource_id, const struct timespec *period, bool *accepted)
- int frsh_rtep_negotiation_messages_vres_get_period (const frsh_resource_id_t resource_id, struct timespec *period)
- int frsh_rtep_service_thread_vres_renegotiate (const frsh_resource_id_t resource_id, const struct timespec *budget, const struct timespec *period, bool *accepted)
- int frsh_rtep_service_thread_vres_get_budget_and_period (const frsh_resource_id_t resource_id, struct timespec *budget, struct timespec *period)

3.9.1 Detailed Description

These functions are needed to set/get the negotiation service parameters.

3.9.2 Function Documentation

3.9.2.1 int frsh_rtep_negotiation_messages_vres_get_period (const frsh_resource_id_t *resource_id*, struct timespec * *period*)

frsh_XXXX_negotiation_messages_vres_get_period()

This function gets the minimum period of the negotiation messages sent through the network.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

→ ***period*** The period for negotiation messages

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.9.2.2 int frsh_rtep_negotiation_messages_vres_renegotiate (const frsh_resource_id_t *resource_id*, const struct timespec * *period*, bool * *accepted*)

frsh_XXXX_negotiation_messages__vres_renegotiate()

This function allows the application to change the minimum period of the negotiation messages sent through the network. It is similar to the service thread but for the network messages. We do not provide budget here because the size of the negotiation messages is fixed.

This change is similar to a renegotiation so a schedulability test must be done to see if the change can be accepted or not.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***period*** The new period for negotiation messages

→ ***accepted*** If the change has been accepted or not

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.9.2.3 int frsh_rtep_service_thread_vres_get_budget_and_period (const frsh_resource_id_t *resource_id*, struct timespec * *budget*, struct timespec * *period*)

frsh_XXXX_service_thread_vres_get_budget_and_period()

This function gets the budget and period of a service thread.

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

→ ***budget*** The budget of the service thread

→ ***period*** The period of the service thread

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

3.9.2.4 int frsh_rtep_service_thread_vres_renegotiate (const frsh_resource_id_t *resource_id*, const struct timespec * *budget*, const struct timespec * *period*, bool * *accepted*)

frsh_XXXX_service_thread_vres_renegotiate()

This function allows the application to change the period and budget of the service thread that makes the negotiations and schedulability tests in a network.

The service thread starts with a default budget and period that should be configurable

Parameters:

← ***resource_id*** The network we are referring to (a protocol could be able to handle several networks at the same time)

← ***budget*** The new budget for the service thread

← ***period*** The new period for the service thread

→ ***accepted*** If the negotiation has been accepted or not

Returns:

FNA_NO_ERROR: in this case it also means contract accepted
FNA_ERR_INTERNAL_ERROR: protocol dependent internal errors
FNA_ERR_NOT_INITIALIZED: if the protocol is not initialized
FNA_ERR_RESOURCE_ID_INVALID: if we are not in charge of resource_id
FNA_ERR_BAD_ARGUMENT: if pointers are NULL

Chapter 4

fna File Documentation

4.1 fna/fna.h File Reference

```
#include "frsh_core_types.h"
#include "fna_error.h"
#include <stdint.h>
```

Typedefs

- typedef void * fna_vres_id_t

Functions

- int fna_init (const frsh_resource_id_t resource_id)
- int fna_vres_create (const frsh_resource_id_t resource_id, const frsh_contract_t *contract, fna_vres_id_t *vres)
- int fna_vres_renegotiate_sync (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, const frsh_contract_t *new_contract)
- int fna_vres_renegotiate_async (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, const frsh_contract_t *new_contract, frsh_signal_t signal_to_notify, frsh_signal_info_t signal_info)
- int fna_vres_get_renegotiation_status (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, frsh_renegotiation_status_t *renegotiation_status)
- int fna_vres_destroy (const frsh_resource_id_t resource_id, const fna_vres_id_t vres)
- int fna_vres_get_contract (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, frsh_contract_t *contract)
- int fna_vres_get_usage (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, struct timespec *usage)
- int fna_vres_get_remaining_budget (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, struct timespec *remaining_budget)
- int fna_vres_get_budget_and_period (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, struct timespec *budget, struct timespec *period)
- int fna_spare_get_capacity (const frsh_resource_id_t resource_id, const int importance, uint32_t *capacity)
- int fna_spare_get_total_weight (const frsh_resource_id_t resource_id, const int importance, int *total_weight)

- int fna_spare_release_capacity (const frsh_resource_id_t resource_id, const fna_vres_id_t vres, const struct timespec new_budget, const struct timespec new_period)
- int fna_send_sync (const frsh_send_endpoint_t *endpoint, const void *msg, const size_t size)
- int fna_send_async (const frsh_send_endpoint_t *endpoint, const void *msg, const size_t size)
- int fna_receive_sync (const frsh_receive_endpoint_t *endpoint, void *buffer, const size_t buffer_size, size_t *received_bytes)
- int fna_receive_async (const frsh_receive_endpoint_t *endpoint, void *buffer, const size_t buffer_size, size_t *received_bytes)
- int fna_send_endpoint_get_pending_messages (const frsh_send_endpoint_t *endpoint, int *number_of_pending_messages)
- int fna_receive_endpoint_get_pending_messages (const frsh_receive_endpoint_t *endpoint, int *number_of_pending_messages)
- int fna_message_get_tx_time (const frsh_resource_id_t resource_id, const size_t nbytes, struct timespec *tx_time)
- int fna_message_get_max_size (const frsh_resource_id_t *resource_id, size_t *max_size)

4.2 fna/frsh_fna.h File Reference

Defines

- #define RTEP

Functions

- int frsh_rtep_map_network_address (const frsh_resource_id_t resource_id, const rtep_network_address_t *in_address, frsh_network_address_t *out_address)
- int frsh_rtep_map_stream_id (const frsh_resource_id_t resource_id, const rtep_channel_t *in_stream, frsh_stream_id_t *out_stream)
- int frsh_rtep_negotiation_messages_vres_renegotiate (const frsh_resource_id_t resource_id, const struct timespec *period, bool *accepted)
- int frsh_rtep_negotiation_messages_vres_get_period (const frsh_resource_id_t resource_id, struct timespec *period)
- int frsh_rtep_service_thread_vres_renegotiate (const frsh_resource_id_t resource_id, const struct timespec *budget, const struct timespec *period, bool *accepted)
- int frsh_rtep_service_thread_vres_get_budget_and_period (const frsh_resource_id_t resource_id, struct timespec *budget, struct timespec *period)

4.2.1 Define Documentation

4.2.1.1 #define RTEP

Definition at line 75 of file frsh_fna.h.

Index