

NORM Developer's Guide

Background

This document describes an application programming interface (API) for the Nack-Oriented Reliable Multicast (NORM) protocol implementation developed by the United States Naval Research Laboratory (NRL). The NORM protocol provides reliable data transport for applications wishing to use Internet Protocol (IP) Multicast services for group data delivery. NORM can also support unicast (point-to-point) data communication and may be used for such when deemed appropriate.

Overview

The NORM API has been designed to provide simple, straightforward access to and control of NORM protocol state and functions. Functions are provided to create and initialize instances of the NORM API and associated transport sessions (*NormSessions*). Subsequently, NORM data transmission (*NormSender*) operation can be activated and the application can queue various types of data (*NormObjects*) for reliable transport. Additionally or alternatively, NORM reception (*NormReceiver*) operation can also be enabled on a per-session basis and the protocol implementation alerts the application of receive events.

By default, the NORM API will create an operating system thread in which the NORM protocol engine runs. This allows user application code and the underlying NORM code to execute somewhat independently of one another. The NORM protocol thread notifies the application of various protocol events through a thread-safe event dispatching mechanism and API calls are provided to allow the application to control NORM operation. (*Note: API mechanisms for lower-level, non-threaded control and execution of the NORM protocol engine will also be provided in the future.*)

The NORM API operation can be summarized by the following categories of functions:

- 1) API Initialization
- 2) Session Creation and Control
- 3) Data Transmission
- 4) Data Reception
- 5) Event Notification

Note the order of these categories roughly reflects the order of function calls required to use NORM in an application. The first step is to create and initialize, as needed, at least one instance of the NORM API. Then one or more NORM transport sessions (where a "session" corresponds to data exchanges on a given multicast group and host port number) may be created and controlled.

(TBD - Address each of the areas, providing links to appropriate portions of the API Reference as example operations and function calls are cited.)

API Reference

API Variable Types and Constants

The NORM API defines and enumerates a number of supporting variable types and values which are used in different function calls. The variable types are described here.

NormInstanceHandle

The `NormInstanceHandle` type is returned when a NORM API instance is created (see `NormCreateInstance()`). This handle can be subsequently used for API calls which require reference to a specific NORM API instance. By default, each NORM API instance instantiated creates an operating system thread for protocol operation. Note that multiple NORM transport sessions may be created for a single API instance. In general, it is expected that applications will create a single NORM API instance, but some multi-threaded application designs may prefer multiple corresponding NORM API instances. The value `NORM_INSTANCE_INVALID` corresponds to an invalid API instance.

NormSessionHandle

The `NormSessionHandle` type is used to reference NORM transport sessions which have been created using the `NormCreateSession()` API call. Multiple `NormSessionHandles` may be associated with a given `NormInstanceHandle`. The special value `NORM_SESSION_INVALID` is used to refer to invalid session references.

NormNodeHandle

The `NormNodeHandle` type is used to reference state kept by the NORM implementation with respect to other participants within a *NormSession*. Most typically, the `NormNodeHandle` is used by receiver applications to dereference information about remote senders of data as needed. The special value `NORM_NODE_INVALID` corresponds to an invalid reference.

NormNodeId

The `NormNodeId` type corresponds to a 32-bit numeric value which should uniquely identify a participant (node) in a given *NormSession*. The `NormNodeGetId()` function can be used to retrieve this value given a valid `NormNodeHandle`. The special value `NORM_NODE_NONE` corresponds to an invalid (or null) node while the value `NORM_NODE_ANY` serves as a wildcard value for some functions.

NormObjectHandle

The `NormObjectHandle` type is used to reference state kept for data transport objects being actively transmitted or received. The state kept for NORM transport objects is temporary, but the NORM API provides a function to persistently retain state associated with a sender or receiver `NormObjectHandle` (see `NormObjectRetain()`) if needed. For sender objects, unless explicitly retained, the `NormObjectHandle` can be

considered valid until the referenced object is explicitly canceled (see `NormObjectCancel()`) or purged from the sender transmission queue (see the event `NORM_TX_OBJECT_PURGED`). For receiver objects, these handles should be treated as valid only until a subsequent call to `NormGetNextEvent()` unless, again, specifically retained. The special value `NORM_OBJECT_INVALID` corresponds to an invalid transport object reference.

NormObjectType

The `NormObjectType` type is an enumeration of possible NORM data transport object types. As previously mentioned, valid types include:

- 1) `NORM_OBJECT_FILE`
- 2) `NORM_OBJECT_DATA`, and
- 3) `NORM_OBJECT_STREAM`

Given a `NormObjectHandle`, the application may determine an object's type using the `NormObjectGetType()` function call. A special `NormObjectType` value, `NORM_OBJECT_NONE`, indicates an invalid object type.

NormObjectTransportId

The `NormObjectTransportId` type is a 16-bit numerical value assigned to *NormObjects* by senders during active transport. These values are temporarily unique with respect to a given sender within a *NormSession* and may be "recycled" for use for future transport objects. NORM sender nodes assign these values in a monotonically increasing fashion during the course of a session as part of protocol operation. Typically, the application should not need access to these values, but an API call `NormObjectGetTransportId()` is provided to retrieve these values if needed. *(Note this function may be deprecated – it may not be needed at all if the `NormObjectRequeue()` function (TBD) is implemented using handles only, but some applications requiring persistence even after a system reboot may need the ability to recall previous transport ids?)*

NormEventType

The `NormEventType` is an enumeration of NORM API events. "Events" are used by the NORM API to signal the application of significant NORM protocol operation events (e.g., receipt of a new receive object, etc). A description of possible `NormEventType` values and their interpretation is given below. The function call `NormGetNextEvent()` is used to retrieve events from the NORM protocol engine.

NormEvent

The `NormEvent` type is a structure used to describe significant NORM protocol events. This structure is defined as follows:

```
typedef struct
{
    NormEventType    type;
```

```
    NormSessionHandle session;
    NormNodeHandle     node;
    NormObjectHandle   object;
} NormEvent;
```

The `type` field indicates the `NormEventType` and determines how the other fields should be interpreted. Note that not all `NormEventType` fields are relevant to all events. The `session`, `node`, and `object` fields indicate the applicable `NormSessionHandle`, `NormNodeHandle`, and `NormObjectHandle`, respectively, to which the event applies. NORM protocol events are made available to the application via the `NormGetNextEvent()` function call.

API Initialization

The first step in using the NORM API is to create an "instance" of the NORM protocol engine. Note that multiple instances may be created by the application if necessary, but generally only a single instance is required since multiple *NormSessions* may be managed under a single NORM API instance.

NormCreateInstance()

Synopsis

```
#include <normApi.h>
NormInstanceHandle NormCreateInstance();
```

Description

This function creates an instance of a NORM protocol engine and is the necessary first step before any other API functions may be used. With the instantiation of the NORM protocol engine, an operating system thread is created for protocol execution. The returned `NormInstanceHandle` value may be used in subsequent API calls as needed, such as `NormCreateSession()`, etc.

Return Values

A value of `NORM_INSTANCE_INVALID` is returned upon failure. The function will only fail if system resources are unavailable to allocate the instance and/or create the corresponding thread.

NormDestroyInstance()

Synopsis

```
#include <normApi.h>
void NormDestroyInstance(NormInstanceHandle instance);
```

Description

The `NormDestroyInstance()` function immediately shuts down and destroys the NORM protocol engine instance referred to by the instance parameter. The

application should make no subsequent references to the indicated `NormInstanceHandle` or any other API handles or objects associated with it. However, the application is still responsible for releasing any object handles it has retained (see `NormObjectRetain()` and `NormObjectRelease()`).

Return Values

The function has no return value.

NormSetCacheDirectory()

Synopsis

```
#include <normApi.h>
bool NormSetCacheDirectory(NormInstanceHandle instance,
                           const char* cachePath);
```

Description

This function sets the directory path used by receivers to cache newly-received `NORM_OBJECT_FILE` objects. This function must be called before any file objects may be received and thus should be called before any calls to `NormStartReceiver()` are made. However, note that the cache directory may be changed even during active NORM reception. In this case, the new specified directory path will be used for subsequently-received files. Any files received before a directory path change will remain in the previous cache location. Note that the `NormFileRename()` function may be used to rename, and thus potentially move, received files after reception has begun.

The `instance` parameter specifies the NORM protocol engine instance (all `NormSessions` associated with that `instance` share the same cache path) and the `cachePath` is a string specifying a valid (and writable) directory path. The function returns *true* on success and *false* on failure. The failure conditions are that the indicated directory does not exist or the process does not have permissions to write.

NormGetNextEvent()

Synopsis

```
#include <normApi.h>
bool NormGetNextEvent(NormInstanceHandle instance,
                      NormEvent* theEvent);
```

Description

This function retrieves the next available NORM protocol event from the protocol engine. The `instance` parameter specifies the applicable NORM protocol engine, and the `theEvent` parameter must be a valid pointer to a `NormEvent` structure capable of

receiving the NORM event information. An enumeration of the types of NORM protocol events and their intended interpretation is provided later.

Note that this is currently the only blocking call in the NORM API. But non-blocking operation may be achieved by using the `NormGetDescriptor()` function to obtain a descriptor (or HANDLE for WIN32) suitable for asynchronous input/output (I/O) notification using such system calls as `select()` (UNIX) or `WaitForMultipleObjects()` (WIN32). The descriptor is signaled when an event is pending.

Return Values

This function generally blocks the thread of application execution until a `NormEvent` is available and returns `true` when a `NormEvent` is available. However, there are some cases when the function may return even when no event is pending. In these cases, the return value is `false`.

WIN32 Note: A future version of this API will provide an option to have a user-defined Window message posted when a NORM API event is pending. (Also some event filtering calls may be provided (e.g. avoid the potentially numerous RX_OBJECT_UPDATED events if undesired)).

NormGetDescriptor()

Synopsis

```
#include <normApi.h>
NormDescriptor NormGetDescriptor(NormInstanceHandle instance);
```

Description

This function is used to retrieve a `NormDescriptor` (integer file descriptor (UNIX) or HANDLE (WIN32)) suitable for asynchronous I/O notification to avoid blocking calls to `NormGetNextEvent()`. A `NormDescriptor` is available for each protocol engine instance. The descriptor (or WIN32 HANDLE) is suitable for use as an input (or "read") descriptor which is signaled when a NORM protocol event is ready for retrieval via `NormGetNextEvent()`. Hence, a call to `NormGetNextEvent()` will not block when the descriptor has been signaled. The `select()` system call (UNIX) (or `WaitForMultipleObjects()` (WIN32)) can be used to detect when the returned `NormDescriptor` is signaled. For the `select()` call usage, the NORM descriptor should be treated as a "read" descriptor.

Return Values

A descriptor is returned which is valid until a call to `NormDestroyInstance()` is made. Upon error, a value of `NORM_DESCRIPTOR_INVALID` is returned.

NORM Session Creation and Control

NormCreateSession()

Synopsis

```
#include <normApi.h>
```

```
NormSessionHandle NormCreateSession(NormInstanceHandle    instance,  
                                     const char*          address,  
                                     unsigned short       port,  
                                     NormNodeId           localId);
```

Description

This function creates a NORM reliable multicast session (*NormSession*) using the address parameters provided. While session state is allocated and initialized, active session participation does not begin until a call is made to `NormStartSender()` and/or `NormStartReceiver()` to join the specified multicast group (if applicable) and start protocol operation. The following parameters are required in this function call:

- | | |
|-----------------------|---|
| <code>instance</code> | This must be a valid <code>NormInstanceHandle</code> previously obtained with a call to <code>NormCreateInstance()</code> . |
| <code>address</code> | This points to a string containing an IP address (e.g. dotted decimal IPv4 address (or IPv6 address) or name resolvable to a valid IP address. The specified <u>address</u> (along with the <u>port</u> number) determines the destination of NORM messages sent. For multicast sessions, NORM senders and receivers must use a common multicast address and port number. For unicast sessions, the sender and receiver must use a common port number, but specify the other node's IP address as the session address (Although note that receiver-only unicast nodes who are providing unicast feedback to senders will not generate any messages to the session IP address and the <u>address</u> parameter value is thus inconsequential for this special case). |
| <code>port</code> | This must be a valid, unused port number corresponding to the desired NORM session address. See the <u>address</u> parameter description for more details. |
| <code>localId</code> | The <code>localId</code> parameter specifies the <code>NormNodeId</code> that should be used to identify the application's presence in the <i>NormSession</i> . All participant's in a <i>NormSession</i> should use unique <code>localId</code> values. The application may specify a value of <code>NORM_NODE_ANY</code> or <code>NORM_NODE_ANY</code> for the <code>localId</code> parameter. In this case, the NORM implementation will attempt to pick an identifier based on the host computer's "default" IP address (based on the computer's default host name). Note there is a chance that this approach may not provide unique node identifiers in some situations and the NORM protocol does not currently provide a mechanism to detect or resolve <i>NormNodeId</i> collisions. Thus, the application should explicitly specify the |

localId unless there is a high degree of confidence that the default IP address will provide a unique identifier.

Return Values

The returned NormSessionHandle value is valid until a call to NormDestroySession() is made. A value of *NORM_SESSION_INVALID* is returned upon error.

NormDestroySession()

Synopsis

```
#include <normApi.h>
void NormDestroySession(NormSessionHandle session);
```

Description

This function immediately terminates the application's participation in the *NormSession* identified by the session parameter and frees any resources used by that session. An exception to this is that the application is responsible for releasing any explicitly retained NormObjectHandles (See NormObjectRetain() and NormObjectRelease()).

Return Values

This function has no returned values.

NormStartSender()

Synopsis

```
#include <normApi.h>
bool NormStartSender(NormSessionHandle session
                    unsigned long    bufferSize
                    unsigned short   segmentSize,
                    unsigned char    blockSize,
                    unsigned char    numParity);
```

Description

The application's participation as a sender within a specified *NormSession* begins when this function is called. This includes protocol activity such as congestion control and/or group round-trip timing (GRTT) feedback collection and application API activity such as posting of sender-related *NormEvents*. The parameters required for this function call include:

session This must be a valid NormSessionHandle previously obtained with a call to NormCreateSession().

bufferSpace This specifies the maximum memory space the NORM protocol

engine is allowed to use to buffer any sender calculated FEC segments and repair state for the session. The optimum bufferSpace value is function of the network topology *bandwidth*delay* product and packet loss characteristics. If the bufferSpace limit is too small, the protocol may operate less efficiently as the sender is required to possibly recalculate FEC parity segments and/or provide less efficient repair transmission strategies (resort to explicit repair) when state is dropped due to constrained buffering resources. However, note the protocol will still provide reliable transfer. A large bufferSpace allocation is safer at the expense of possibly committing more memory resources.

segmentSize This parameter sets the maximum *payload* size (in bytes) of NORM sender messages (*not* including any NORM message header fields). A sender's segmentSize value is also used by receivers to limit the payload content of some feedback messages (e.g. NORM_NACK message content, etc.) generated in response to that sender. Note different senders within a *NormSession* may use different segmentSize values. Generally, the appropriate segment size to use is dependent upon the types of networks forming the multicast topology, but applications may choose different values for other purposes. Note that application designers **MUST** account for the size of NORM message headers when selecting a segmentSize. For example, the NORM_DATA message header for a *NORM_OBJECT_STREAM* with full header extensions is 48 bytes in length. In this case, the UDP payload size of these messages generated by NORM would be up to $(48 + \text{segmentSize})$ bytes.

blockSize This parameter sets the number of source symbol segments (packets) per coding block, for the systematic Reed-Solomon FEC code used in the current NORM implementation. For traditional systematic block code " (n,k) " nomenclature, the blockSize value corresponds to $(n-k)$. NORM logically segments transport object data content into coding blocks and the blockSize parameter determines the number of source symbol segments (packets) comprising a single coding block where each source symbol segment is up to segmentSize bytes in length.. A given block's parity symbol segments are calculated using the corresponding set of source symbol segments. The maximum blockSize allowed by the 8-bit Reed-Solomon codes in NORM is 255, with the further limitation that $(\text{blockSize} + \text{numParity}) \leq 255$.

numParity This parameter sets the maximum number of parity symbol segments (packets) the sender is willing to *calculate* per FEC coding block. The parity symbol segments for a block are calculated from the corresponding blockSize source symbol segments. In the " (n,k) " nomenclature mention above, the numParity value corresponds to " k ". A property of the Reed-Solomon FEC codes used in the current

NORM implementation is that one parity segment can fill any one erasure (missing segment (packet)) for a coding block. For a given `blockSize`, the maximum `numParity` value is $(255 - \text{blockSize})$. However, note that computational complexity increases significantly with increasing `numParity` values and applications may wish to be conservative with respect to `numParity` selection, given anticipated network packet loss conditions and group size scalability concerns. Additional FEC code options may be provided for this NORM implementation in the future with different parameters, capabilities, trade-offs, and computational requirements.

These parameters are currently immutable with respect to a sender's participation within a *NormSession*. Sender operation must be stopped (see `NormStopSender()`) and restarted with another call to `NormStartSender()` if these parameters require alteration. The API may be extended in the future to support additional flexibility here, if required. For example, the NORM protocol "*sessionId*" field may possibly be leveraged to permit a node to establish multiple virtual presences as a sender within a *NormSession* in the future. This would allow the sender to provide multiple concurrent streams of transport, with possibly different FEC and other parameters if appropriate within the context of a single *NormSession*. Again, this extended functionality is not yet supported in this implementation.

Return Values

A value of *true* is returned upon success and *false* upon failure. The reasons failure may occur include limited system resources or that the network sockets required for communication failed to open or properly configure. *(TBD – Provide a NormGetError(NormSessionHandle session) function to retrieve a more specific error indication for this and other functions.)*

NormStopSender()

Synopsis

```
#include <normApi.h>

void NormStopSender(NormSessionHandle session,
                   bool graceful = false);
```

Description

This function terminates the application's participation in a *NormSession* as a sender. By default, the sender will immediately exit the session without notifying the receiver set of its intention. However a "graceful shutdown" option is provided to terminate sender operation gracefully, notifying the receiver set its pending exit with appropriate protocol messaging. A *NormEvent*, `NORM_LOCAL_SERVER_CLOSED`, is dispatched when the graceful shutdown process has completed.

Description

This function ends the application's participation as a receiver in the *NormSession* specified by the `session` parameter. By default, all receiver-related protocol activity is immediately halted and all receiver-related resources are freed (except for those which have been specifically retained (see `NormObjectRetain()`). However, and optional `gracePeriod` parameter is provided to allow the receiver an opportunity to inform the group of its intention. This is applicable when the local receiving *NormNode* has been designated as an active congestion control representative (i.e. current limiting receiver (CLR) or potential limiting receiver (PLR)). In this case, a non-zero `gracePeriod` value provides an opportunity for the receiver to respond to the applicable sender(s) so the sender will not expect further congestion control feedback from this receiver. The `gracePeriod` integer value is used as a multiplier with the largest sender GRTT to determine the actual time period for which the receiver will linger in the group to provide such feedback (i.e. "grace time" = (`gracePeriod` * *GRTT*)). During this time, the receiver will not generate any requests for repair or other protocol actions aside from response to applicable congestion control probes. When the receiver is removed from the current list of receivers in the sender congestion control probe messages (or the `gracePeriod` expires, whichever comes first), the NORM protocol engine will post a `NORM_LOCAL_RECEIVER_CLOSED` event for the applicable `session`, and related resources are freed.

Return Values

This function has no return values.

NormSetTransmitRate()

Synopsis

```
#include <normApi.h>

void NormSetTransmitRate(NormSessionHandle session,
                        double rate);
```

Description

This function sets the transmission rate limit (in bits per second (bps)) used for NormSender transmissions. For fixed-rate transmission of `NORM_OBJECT_FILE` or `NORM_OBJECT_DATA`, this limit determines the data rate at which NORM protocol messages and data content. For `NORM_OBJECT_STREAM` transmissions, this is the maximum rate allowed for transmission. Note that the application will need to consider the overhead of NORM protocol headers when determining an appropriate transmission rate for its purposes. When NORM congestion control is enabled (see `NormSetCongestionControl()`), the rate set here will be set, but congestion control operation may quickly readjust the rate unless disabled.

Return Values

This function has no return values.

NormSetTransmitRateBounds ()

Synopsis

```
#include <normApi.h>

bool NormSetTransmitRateBounds(NormSessionHandle session,
                               double rateMin,
                               double rateMax);
```

Description

This function sets the range of sender transmission rates within which the NORM congestion control algorithm is allowed to operate. By default, the NORM congestion control algorithm operates with no lower or upper bound on its rate adjustment. This function allows this to be limited where `rateMin` corresponds to the minimum transmission rate (bps) and `rateMax` corresponds to the maximum transmission rate. One or both of these parameters may be set to values less than zero to remove one or both bounds. For example `"NormSetTransmitRate(session, -1.0, 64000.0)"` will set an upper limit of 64 kbps for the sender transmission rate with no lower bound. These rate bounds apply only when congestion control operation is enabled (see `NormSetCongestionControl ()`). If the current congestion control rate falls outside of the specified bounds, the sender transmission rate will

Return Values

This function returns *true* upon success. If (`rateMax` < `rateMin`), the rate bounds will remain unset or unchanged and the function will return *false*.

NormSetGrttEstimate ()

Synopsis

```
#include <normApi.h>

void NormSetGrttEstimate(NormSessionHandle session,
                        double grtt);
```

Description

This function sets the sender's estimate of group round-trip timing (GRTT). This function is expected to most typically used to initialize the sender's GRTT estimate prior to the call to `NormStartSender ()` when the application has a priori confidence that the default initial GRTT value of 0.5 second is inappropriate. The sender GRTT estimate will be updated during normal sender protocol operation after sender startup or if this call is made while sender operation is active. For experimental purposes (or very special application needs), this API provides a mechanism to control or disable the sender GRTT update process (see `NormSetGrttProbing ()`). The `grtt` value will be limited to the maximum GRTT as set (see `NormSetGrttMax ()`) or the default maximum of 10 seconds.

The sender GRTT is advertised to the receiver group and is used to scale various NORM protocol timers. The default NORM GRTT estimation process dynamically measures round-trip timing to determine an appropriate operating value. An overly-large GRTT estimate can introduce additional latency into the reliability process (resulting in a larger virtual *delay*bandwidth* product for the protocol and potentially requiring more buffer space to maintain reliability). An overly-small GRTT estimate may introduce the potential for feedback implosion, limiting the scalability of group size.

Return Values

This function has no return values.

NormGetLocalNodeId()

Synopsis

```
#include <normApi.h>
NormNodeId NormGetLocalNodeId(NormSessionHandle session);
```

Description

This function retrieves the `NormNodeId` value used for the application's participation in the *NormSession* identified by the `session` parameter. The value may have been explicitly set during the `NormCreateSession()` call or derived using the host computer's "default" IP network address.

Return Values

The returned value indicates the *NormNode* identifier used by the NORM protocol engine for the application's participation in the specified *NormSession*.

NormSetMulticastInterface()

Synopsis

```
#include <normApi.h>
bool NormSetMulticastInterface(NormSessionHandle session,
                               const char*         interfaceName);
```

Description

This function specifies which host network interface is used for IP Multicast transmissions and group membership. This generally should be called *before* any call to `NormStartSender()` or `NormStartReceiver()` is made. However, if a call to `NormSetMulticastInterface()` is made *after* either of these function calls, the call will not affect the group membership interface, but only dictate that a possibly different network interface is used for transmitted NORM messages. Thus, the code:

```
NormSetMulticastInterface(session, "interface1");
NormStartReceiver(session, ...);
NormSetMulticastInterface(session, "interface2");
```

will result in NORM group membership (i.e. multicast reception) being managed on "interface1" while NORM multicast transmissions are made via "interface2".

Return Values

A return value of true indicates success while a return value of false indicates that the specified interface was valid. This function will always return true if made before calls to `NormStartSender()` or `NormStartReceiver()`. However, those calls may fail if an invalid interface is specified.

NormSetTTL()

NormSetTOS()

NormSetLoopback()

NormAddAckingNode()

NormRemoveAckingNode()

NormSetWatermark()

NORM Data Transport

The NORM protocol supports transport of three basic types of data content. These include the types *NORM_OBJECT_FILE* and *NORM_OBJECT_DATA* which represent predetermined, fixed-size application data content. The only differentiation with respect to these two types is the implicit “hint” to the receiver to use non-volatile (i.e. file system) storage or memory. This “hint” lets the receiver allocate appropriate storage space with no other information on the incoming data. The NORM implementation reads/writes data for the *NORM_OBJECT_FILE* type directly from/to file storage, while application memory space is accessed for the *NORM_OBJECT_DATA* type. The third data content type, *NORM_OBJECT_STREAM*, represents unbounded, possibly persistent, streams of data content. Using this transport paradigm, traditional, byte-oriented streaming transport service (e.g. similar to that provided by a TCP socket) can be provided. Additionally, NORM has provisions for application-defined message-oriented transport where receivers can recover message boundaries without any “handshake” with the sender. Stream content is buffered by the NORM implementation for transmission/retransmission and as it is received.

The behavior of data transport operation is largely placed in the control of the NORM sender(s). NORM senders controls their data transmission rate, forward error correction (FEC) encoding settings, and parameters controlling feedback from the receiver group. Multiple senders may operate in a session, each with independent transmission parameters. NORM receivers learn needed parameter values from fields in NORM message headers.

NORM transport “objects” (file, data, or stream) are queued for transmission by NORM senders. NORM senders may also cancel transmission of objects at any time. The NORM sender controls the transmission rate either manually (fixed transmission rate) or automatically when NORM congestion control operation is enabled. The NORM congestion control mechanism is designed to be "friendly" to other data flows on the network, fairly sharing available bandwidth.

The NRL NORM implementation also supports optional collection of positive acknowledgment from a subset of the receiver group at application-determined positions during data transmission. The NORM API allows the application to specify the receiver subset ("acking node list") and set "watermark" points for which positive acknowledgement is collected. This process can provide the application with flow control for a critical set of receivers in the group.

In the case of *NORM_OBJECT_FILE* and *NORM_OBJECT_DATA* objects, each NORM transport

and NORM receivers are able to identify these objects by sender-assigned “NormTransportId” values.

NORM Object Functions

NormObjectGetType()

NormObjectGetInfo()

NormObjectGetTransportId()

NormObjectSetNackingMode()

NormObjectCancel()

NormObjectRetain()

NormObjectRelease()

NORM_OBJECT_FILE Transport

NormFileEnqueue()

NormSetCacheDirectory()

NormFileGetName()

NormFileRename()

NORM_OBJECT_DATA Transport

NormDataEnqueue()

NORM_OBJECT_STREAM Transport

NormStreamOpen()

NormStreamClose()

NormStreamSetFlushMode()

NormStreamSetPushMode()

NormStreamWrite()

NormStreamMarkEom()

NormStreamFlush()

NormStreamRead()

NormStreamSeekMsgStart ()