NORM Developer's Guide (version 1.5b5)

Abstract

This document describes an application programming interface (API) for the Nack-Oriented Reliable Multicast (NORM) protocol implementation developed by the Protocol Engineering and Advance Networking (PROTEAN) Research Group of the United States Naval Research Laboratory (NRL). The NORM protocol provides general purpose 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. The current NORM protocol specification is given in the Internet Engineering Task Force (IETF) RFC 3940. This document is currently a reference guide to the NORM API of the NRL reference implementation. More tutorial material may be include in a future version of this document or a separate developer's tutorial may be created at a later date.

Table of Contents

1. Background 3
2. Overview 4
    2.1. API Initialization 5
    2.2. Session Creation and Control 5
    2.3. Data Transport 5
        2.3.1. Data Transmission 5
        2.3.2. Data Reception 7
    2.4. API Event Notification 7
3. Build Notes 7
    3.1. Unix Platforms 8
    3.2. Win32/WiNCE Platforms 8
4. API Reference 8
    4.1. API Variable Types and Constants 8
        4.1.1. NormInstanceHandle 9
        4.1.2. NormSessionHandle 9
        4.1.3. NormSessionId 9
        4.1.4. NormNodeHandle 9
        4.1.5. NormNodeId 9
        4.1.6. NormObjectHandle 9
        4.1.7. NormObjectType 9
        4.1.8. NormSize 10
        4.1.9. NormObjectTransportId 10
        4.1.10. NormEventType 10
        4.1.11. NormEvent 10
        4.1.12. NormDescriptor 10
        4.1.13. NormFlushMode 10
        4.1.14. NormProbingMode 11
        4.1.15. NormSyncPolicy 11
        4.1.16. NormNackingMode 11
        4.1.17. NormRepairBoundary 11
        4.1.18. NormAckingStatus 12
    4.2. API Initialization and Operation 12

4.2.1. NormCreateInstance() 12
4.2.2. NormDestroyInstance() 12
4.2.3. NormStopInstance() 13
4.2.4. NormRestartInstance() 13
4.2.5. NormSetCacheDirectory() 13
4.2.6. NormGetNextEvent() 14
4.2.7. NormGetDescriptor() 16
4.3. Session Creation and Control Functions 17
4.3.1. NormCreateSession() 17
4.3.2. NormDestroySession() 18
4.3.3. NormSetUserData() 18
4.3.4. NormGetUserData() 18
4.3.5. NormGetLocalNodeId() 19
4.3.6. NormSetTxPort() 19
4.3.7. NormSetTxOnly() 20
4.3.8. NormSetRxPortReuse() 20
4.3.9. NormSetMulticastInterface() 22
4.3.10. NormSetSSM() 22
4.3.11. NormSetTTL() 22
4.3.12. NormSetTOS() 22
4.3.13. NormSetLoopback() 23
4.3.14. NormSetFragmentation() 23
4.4. NORM Sender Functions 24
4.4.1. NormStartSender() 24
4.4.2. NormStopSender() 25
4.4.3. NormSetTxRate() 26
4.4.4. NormGetTxRate() 26
4.4.5. NormSetTxSocketBuffer() 26
4.4.6. NormSetFlowControl() 27
4.4.7. NormSetCongestionControl() 27
4.4.8. NormSetTxRateBounds() 28
4.4.9. NormSetTxCacheBounds() 29
4.4.10. NormSetAutoParity() 29
4.4.11. NormGetGrttEstimate() 30
4.4.12. NormSetGrttEstimate() 30
4.4.13. NormSetGrttMax() 31
4.4.14. NormSetGrttProbingMode() 31
4.4.15. NormSetGrttProbingInterval() 32
4.4.16. NormSetBackoffFactor() 32
4.4.17. NormSetGroupSize() 33
4.4.18. NormSetTxRobustFactor() 33
4.4.19. NormFileEnqueue() 34
4.4.20. NormDataEnqueue() 35
4.4.21. NormRequeueObject() 35
4.4.22. NormStreamOpen() 36
4.4.23. NormStreamClose() 37
4.4.24. NormStreamWrite() 37
4.4.25. NormStreamFlush() 38
4.4.26. NormStreamSetAutoFlush() 39
4.4.27. NormStreamSetPushEnable() 39
4.4.28. NormStreamHasVacancy() 40
4.4.29. NormStreamMarkEom() 40
4.4.30. NormSetWatermark() 40
4.4.31. NormCancelWatermark() 42
4.4.32. NormAddAckingNode() 42
4.4.33. NormRemoveAckingNode() 42
4.4.34. NormGetNextAckingNode() 43
4.4.35. NormGetAckingStatus() 43

        - 4.4.36. NormSendCommand() 44
        - 4.4.37. NormCancelCommand() 45
    - 4.5. NORM Receiver Functions 45
        - 4.5.1. NormStartReceiver() 45
        - 4.5.2. NormStopReceiver() 46
        - 4.5.3. NormSetRxCacheLimit() 46
        - 4.5.4. NormSetRxSocketBuffer() 47
        - 4.5.5. NormSetSilentReceiver() 47
        - 4.5.6. NormSetDefaultUnicastNack() 48
        - 4.5.7. NormNodeSetUnicastNack() 48
        - 4.5.8. NormSetDefaultSyncPolicy() 49
        - 4.5.9. NormSetDefaultNackingMode() 49
        - 4.5.10. NormNodeSetNackingMode() 50
        - 4.5.11. NormObjectSetNackingMode() 50
        - 4.5.12. NormSetDefaultRepairBoundary() 51
        - 4.5.13. NormNodeSetRepairBoundary() 51
        - 4.5.14. NormSetDefaultRxRobustFactor() 51
        - 4.5.15. NormNodeSetRxRobustFactor() 52
        - 4.5.16. NormStreamRead() 52
        - 4.5.17. NormStreamSeekMsgStart() 53
        - 4.5.18. NormStreamGetReadOffset() 53
    - 4.6. NORM Object Functions 53
        - 4.6.1. NormObjectGetType() 54
        - 4.6.2. NormObjectHasInfo() 54
        - 4.6.3. NormObjectGetInfoLength() 54
        - 4.6.4. NormObjectGetInfo() 55
        - 4.6.5. NormObjectGetSize() 55
        - 4.6.6. NormObjectGetBytesPending() 55
        - 4.6.7. NormObjectCancel() 56
        - 4.6.8. NormObjectRetain() 56
        - 4.6.9. NormObjectRelease() 57
        - 4.6.10. NormFileGetName() 57
        - 4.6.11. NormFileRename() 57
        - 4.6.12. NormDataAccessData() 58
        - 4.6.13. NormDataDetachData() 58
        - 4.6.14. NormObjectGetSender() 59
    - 4.7. NORM Node Functions 59
        - 4.7.1. NormNodeGetId() 59
        - 4.7.2. NormNodeGetAddress() 60
        - 4.7.3. NormNodeGetGrtt() 60
        - 4.7.4. NormNodeGetCommand() 60
        - 4.7.5. NormNodeFreeBuffers() 61
        - 4.7.6. NormNodeDelete() 61
        - 4.7.7. NormNodeRetain() 62
        - 4.7.8. NormNodeRelease() 62
    - 4.8. NORM Debugging Functions 63
        - 4.8.1. NormSetDebugLevel() 63
        - 4.8.2. NormOpenDebugLog() 63
        - 4.8.3. NormCloseDebugLog() 63
        - 4.8.4. NormOpenDebugPipe() 63
        - 4.8.5. NormCloseDebugPipe() 64

1. Background

This document describes an application programming interface (API) for the Nack-Oriented Reliable Multicast (NORM) protocol implementation developed by the Protocol Engineering and Advance Networking (PROTEAN) Research Group of the United States Naval Research Laboratory (NRL). The NORM protocol provides general purpose 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. The current NORM protocol specification is given in the Internet Engineering Task Force (IETF) RFC 5740.

The NORM protocol is designed to provide end-to-end reliable transport of bulk data objects or streams over generic IP multicast routing and forwarding services. NORM uses a selective, negative acknowledgement (NACK) mechanism for transport reliability and offers additional protocol mechanisms to conduct reliable multicast sessions with limited "a priori" coordination among senders and receivers. A congestion control scheme is specified to allow the NORM protocol to fairly share available network bandwidth with other transport protocols such as Transmission Control Protocol (TCP). It is capable of operating with both reciprocal multicast routing among senders and receivers and with asymmetric connectivity (possibly a unicast return path) from the senders to receivers. The protocol offers a number of features to allow different types of applications or possibly other higher-level transport protocols to utilize its service in different ways. The protocol leverages the use of FEC-based repair and other proven reliable multicast transport techniques in its design.

The NRL NORM library attempts to provide a general useful capability for development of reliable multicast applications for bulk file or other data delivery as well as support of stream-based transport with possible real-time delivery requirements. The API allows access to many NORM protocol parameters and control functions to tailor performance for specific applications. While default parameters, where provided, can be useful to a potential wide range of requirements, the many different possible group communication paradigms dictate different needs for different applications. Even with NORM, the developer should have a thorough understanding of the specific application's group communication needs.

2. 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 code may also be provided in the future.)

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

1. API Initialization
2. Session Creation and Control
3. Data Transport
4. API 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 (or unicast address) and host port number) may be created and controlled. Applications may participate as senders and/or receivers within a NORM session. NORM senders transmit data to the session destination address (usually an IP multicast group) while receivers are notified of incoming data. The NORM API provides and event notification scheme to notify the application of significant sender and receiver events. There are also a number support functions provided for the application to control and monitor its participation within a NORM transport session.

2.1. API Initialization

The NORM API requires that an application explicitly create at least one instance of the NORM protocol engine that is subsequently used as a conduit for further NORM API calls. By default, the NORM protocol engine runs in its own operating system thread and interacts with the application in a thread-safe manner through the API calls and event dispatching mechanism.

In general, only a single thread should access the `NormGetNextEvent()` API call for a given *NormInstance*. This function serves as the conduit for delivering NORM protocol engine events to the application. A NORM application can be designed to be single-threaded, even with multiple active NormSessions, but also multiple API instances can be created (see `NormCreateInstance()`) as needed for applications with specific requirements for accessing and controlling participation in multiple *NormSessions* from separate operating system multiple threads. Or, alternatively, a single *NormInstance* could be used, with a "master thread" serving as an intermediary between the `NormGetNextEvent()` function, demultiplexing and dispatching events as appropriate to other "child threads" that are created to handle "per-*NormSession*" input/output. The advantage of this alternative approach is that the end result would be one NORM protocol engine thread plus one "master thread" plus one "child thread" per *NormSession* instead of two threads (protocol engine plus application thread) per *NormSession* if such multi-threaded operation is needed by the application.

2.2. Session Creation and Control

Once an API instance has been successfully created, the application may then create NORM transport session instances as needed. The application can participate in each session as a sender and/or receiver of data. If an application is participating as a sender, it may enqueue data transport objects for transmission. The control of transmission is largely left to the senders and API calls are provided to control transmission rate, FEC parameters, etc. Applications participating as receivers will be notified via the NORM API's event dispatching mechanism of pending and completed reliable reception of data along with other significant events. Additionally, API controls for some optional NORM protocol mechanisms, such as positive acknowledgment collection, are also provided.

Note when multiple senders are involved, receivers allocate system resources (buffer space) for each active sender. With a very large number of concurrently active senders, this may translate to significant memory allocation on receiver nodes. Currently, the API allows the application to control how much buffer space is allocated for each active sender (NOTE: In the future, API functions may be provided limit the number of active senders monitored and/or provide the application with finer control over receive buffer allocation, perhaps on a per sender basis).

2.3. 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.

2.3.1. Data Transmission

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.`NormSetAutoParity()`) to achieve reliable transfer) receive object transmission before any extensive repair process that may be required to satisfy other receivers with poor network connectivity. The repair boundary can also be set for individual remote senders using the `NormNodeSetRepairBoundary()` function.`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.

By default, the NORM sender transmits application-enqueued data content, providing repair transmissions (usually in the form of FEC messages) only when requested by NACKs from the receivers. However, the application may also configure NORM to proactively send some amount of FEC content along with the original data content to create a "robust" transmission that, in some cases, may be reliably received without any NACKing activity. This can allow for some degree of reliable protocol operation even without receiver feedback available. NORM senders may also requeue (within the limits of "transmit cache" settings) objects for repeat transmission, and receivers may combine together multiple transmissions to reliably receive content. Additionally, hybrid proactive/reactive FEC repair operation is possible with the receiver NACK process as a "backup" for when network packet loss exceeds the repair capability of the proactive FEC settings.

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 explicit flow control for an application-determined critical set of receivers in the group.

For a NORM application to perform data transmission, it must first create a session using `NormCreateSession()` and make a call to `NormStartSender()` before sending actual user data. The functions `NormFileEnqueue()`, `NormDataEnqueue()`, and `NormStreamWrite()` are available for the application to pass data to the NORM protocol engine for transmission. Note that to use `NormStreamWrite()`, a "sender stream" must first be created using `NormStreamOpen()`. In the case of `NormFileEnqueue()` and `NormDataEnqueue()`, the NORM protocol engine directly accesses the application file or memory space to refer to the transmitted content and does not make its own copy of this data.

The calls to enqueue transport objects or write to a stream may be called at any time, but the `NORM_TX_QUEUE_EMPTY` and `NORM_TX_QUEUE_VACANCY` notification events (see `NormGetNextEvent()`) provide useful cues for when these functions may be successfully called. Typically, an application might catch both `NORM_TX_QUEUE_EMPTY` and `NORM_TX_QUEUE_VACANCY` event types as cues for enqueuing additional transport objects or writing to a stream. However, an application may choose to cue off of `NORM_TX_QUEUE_EMPTY` only if it wishes to provide the "freshest" data to NORM for transmission. The advantage of additionally using `NORM_TX_QUEUE_VACANCY` is that if the application uses this cue to fill up NORM transport object or stream buffers, it can keep the NORM stream busy sending data and realize the highest possible transmission rate when attempting very high speed communication (Otherwise, the NORM protocol engine may experience some "dead air time" waiting for the application thread to respond to a `NORM_TX_QUEUE_EMPTY` event). Note the sender application can control buffer depths as needed with the `NormSetTxCacheBounds()` and `NormStreamOpen()` calls. Additionally, it is possible for applications to configure the transmit object "cache" (see `NormSetTxCacheBounds()`) and use the `NormRequeueObject()` call (for objects that have not yet received a *`NORM_TX_OBJECT_PURGED`* notification) to effect a sort of "data carousel" operation with repeated transmission of the cached objects. The *`NORM_TX_OBJECT_SENT`* notification can be used a cue to properly control the "requeue" cycle(s).

The NORM implementation provides a form of timer-based flow control that limits how quickly sender applications may enqueue new objects or stream data for transmission. The `NormSetFlowControl()` call is provided to control this behavior, including the option to disable it. This timer-based mechanism is a type of "soft" flow control by allowing receivers "sufficient" time to request repair of pending data the sender has enqueued. A more explicit form of flow control using the optional "watermark flushing" mechanism is described below.

Another cue that can be leveraged by the sender application to determine when it is appropriate to enqueue (or write) additional data for transmission is the `NORM_TX_WATERMARK_COMPLETED` event. This event is posted when the flushing or explicit positive acknowledgment collection process has completed for a "watermark" point in transmission that was set by the sender (see `NormSetWatermark()` and `NormAddAckingNode()`). A list of `NormNodeId` values can be supplied from which explicit acknowledgement is expected and/or the `NormNodeId` `NORM_NODE_NONE` can be set (using `NormAddAckingNode()`) for completion of a NACK-based version of the watermark flushing procedure. This flushing process can be used as a flow control mechanism for NORM applications. Note this is distinct from NORM's congestion control mechanism that, while it provides network-friendly transmission rate control, does guarantee flow control to receiving nodes.`NORM_NODE_NONE` can be set (using `NormAddAckingNode()`) for completion of a NACK-based version of the watermark flushing procedure. This flushing process can be used as a flow control mechanism for NORM applications. Note this is distinct from NORM's congestion control mechanism that, while it provides network-friendly transmission rate control, does guarantee flow control to receiving nodes.

2.3.2. Data Reception

NORM receiver applications learn of active senders and their corresponding pending and completed data transfers, etc via the API event dispatching mechanism. By default, NORM receivers use NACK messages to request repair of transmitted content from the originating sender as needed to achieve reliable transfer. Some API functions are available to provide some additional control over the NACKing behavior, such as initially NACKing for `NORM_INFO` content only or even to the extent of disabling receiver feedback (silent receiver or emission-controlled (EMCON) operation) entirely. Otherwise, the parameters and operation of reliable data transmission are left to sender applications and receivers learn of sender parameters in NORM protocol message headers and are instructed by `NORM_CMD` messages from the sender(s).

With respect to the NORM API, the receiver application is informed of new senders and receive data objects via the the *`NORM_REMOTE_SENDER_NEW`* and *`NORM_RX_OBJECT_NEW`* notifications, respectfully. Additionally, object reception progress is indicated with the *`NORM_RX_OBJECT_UPDATED`* notification and this also serves as an indicator for the *`NORM_OBJECT_STREAM`* type that the receive application should make calls to `NormStreamRead()` to read newly received stream content. NORM sender status is also conveyed via the *`NORM_REMOTE_SENDER_ACTIVE`* and NORM_REMOTE_SENDER_INACTIVE notifications. For example, the receiver application may use the *`NORM_REMOTE_SENDER_INACTIVE`* as a cue to make calls to `NormNodeFreeBuffers()` and/or `NormNodeDelete()` to free memory resources allocated for buffering received content for the given sender. The amount of memory allocated *per sender* is set in the `NormStartReceiver()` call.

2.4. API Event Notification

An asynchronous event dispatching mechanism is provided to notify the application of significant NORM protocol events. The centerpiece of this is the `NormGetNextEvent()` function that can be used to retrieve the next NORM protocol engine event in the form of a `NormEvent` structure. This function will typically block until a `NormEvent` occurs. However, non-blocking operation may be achieved by using the `NormGetDescriptor()` call to get a NormDescriptor (file descriptor) value (Unix int or Win32 HANDLE) suitable for use in a asynchronous I/O monitoring functions such as the Unix `select()` or Win32 `MsgWaitForMultipleObjects()` system calls. The a NormDescriptor will be signaled when a `NormEvent` is available. For Win32 platforms, dispatching of a user-defined Windows message for NORM event notification is also planned for a future update to the NORM API.

3. Build Notes

To build applications that use the NORM library, a path to the "normApi.h" header file must be provided and the linker step needs to reference the NORM library file ("`libnorm.a`" for Unix platforms and "`Norm.lib`" for Win32 platforms). NORM also depends upon the NRL Protean Protocol Prototyping toolkit "Protokit" library (a.k.a "Protolib") (static library files "`libProtokit.a`" for Unix and "`Protokit.lib`" for Win32). Shared or dynamically-linked versions of these libraries may also be built from the NORM source code or provided. Depending upon the platform, some additional library dependencies may be required to support the needs of NORM and/or Protokit. These are described below.

The "makefiles" directory contains Unix Makefiles for various platforms the "win32" and "wince" sub-directories there contain Microsoft Visual C++ (VC++) and Embedded VC++ project files for building the NORM implementation. Additionally, a "waf" (Python-based build tool) build option is supported that can be used to build and install the NORM library code on the supported platforms. Finally, Python and Java bindings to the NORM API are included and "src/python" and "src/java" directories contain the code for these and the "makefiles/java" directory contains Makefiles to build the NORM Java JNI bindings. Note the "waf" tool can also be used to build the Java and Python bindings.

3.1. Unix Platforms

NORM has been built and tested on Linux (various architectures), MacOS (BSD), Solaris, and IRIX (SGI) platforms. The code should be readily portable to other Unix platforms.

To support IPv6 operation, the NORM and the Protokit library must be compiled with the "`HAVE_IPV6`" macro defined. This is default in the NORM and Protokit Makefiles for platforms that support IPv6. It is important that NORM and Protokit be built with this macro defined the same. With NORM, it is recommended that "large file support" options be enabled when possible.

The NORM API uses threading so that the NORM protocol engine may run independent of the application. Thus the "POSIX Threads" library must be included ("-pthread") in the linking step. MacOS/BSD also requires the addition of the "-lresolv" (resolver) library and Solaris requires the dynamic loader, network/socket, and resolver libraries ("-lnsl -lsocket -lresolv") to achieve successful compilation. The Makefiles in the NORM source code distribution are a reference for these requirements. Note that MacOS 9 and earlier are not supported.

Additionally, it is critical that the `_FILE_OFFSET_BITS` macro be consistently defined for the NORM library build and the application build using the library. The distributed NORM Makefiles have `-D_FILE_OFFSET_BITS=64` set in the compilation to enable "large file support". Applications built using NORM should have the same compilation option set to operate correctly (The definition of the `NormSize` type in "`normApi.h`" depends upon this compilation flag).

3.2. Win32/WiNCE Platforms

NORM has been built using Microsoft's Visual C++ (6.0 and .NET) and Embedded VC++ 4.2 environments. In addition to proper macro definitions (e.g., HAVE_IPV6, etc) that are included in the respective "Protokit" and "NORM" project files, it is important that common code generation settings be used when building the NORM application. The NORM and Protokit projects are built with the "Multi-threading DLL" library usage set. The NORM API requires multi-threading support. This is a critical setting and numerous compiler and linker errors will result if this is not properly set for your application project.

NORM and Protokit also depend on the Winsock 2.0 ("`ws2_32.lib`" (or "`ws2.lib`" (WinCE)) and the IP Helper API ("`iphlpapi.lib`") libraries and these must be included in the project "Link" attributes.

An additional note is that a bug in VC++ 6.0 and earlier compilers (includes embedded VC++ 4.x compilers) prevent compilation of Protokit-based code with debugging capabilities enabled. However, this has been resolved in VC++ .NET and is hoped to be resolved in the future for the WinCE build tools.

Operation on Windows NT4 (and perhaps other older Windows operating systems) requires that the compile time macro `WINVER=0x0400` defined. This is because the version of the IP Helper API library (`iphlpapi.lib`) used by *Protolib* (and hence NORM) for this system doesn't support some of the functions defined for this library. This may be related to IPv6 support issues so it may be possible that the Protolib build could be tweaked to provide a single binary executable suitable for IPv4 operation only across a large range of Windows platforms.

4. API Reference

This section provides a reference to the NORM API variable types, constants and functions.

4.1. 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.

4.1.1. 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.

4.1.2. NormSessionHandle

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

4.1.3. NormSessionId

The `NormSessionId` type is used by applications to uniquely identify their instance of participation as a sender within a *NormSession*. This type is a parameter to the `NormStartSender()` function. Robust applications can use different `NormSessionId` values when initiating sender operation so that receivers can discriminate when a sender has terminated and restarted (whether intentional or due to system failure). For example, an application could cache its prior `NormSessionId` value in non-volatile storage which could then be recovered and incremented (for example) upon system restart to produce a new value. The `NormSessionId` value is used for the value of the instance_id field in NORM protocol sender messages (see the NORM protocol specification) and receivers use this field to detect sender restart within a *NormSession*.

4.1.4. 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.

4.1.5. 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 wild card value for some functions.

4.1.6. 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.

4.1.7. 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.

4.1.8. NormSize

The `NormSize` is the type used for *NormObject* size information. For example, the `NormObjectGetSize()` function returns a value of type `NormSize`. The range of `NormSize` values depends upon the operating system and NORM library compilation settings. With "large file support" enabled, as is the case with distributed NORM library "Makefiles", the `NormSize` type is a 64-bit integer. However, some platforms may support only 32-bit object sizes.

4.1.9. 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 such as `NormObjectGetTransportId()` (*TBD*) may be provided to retrieve these values if needed. (Note this type may be deprecated; i.e., it may not be needed at since the `NormRequeueObject()` function is implemented using handles only, but _some_ applications requiring persistence even after a system reboot may need the ability to recall previous transport ids?)

4.1.10. 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.

4.1.11. 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.

4.1.12. NormDescriptor

The `NormDescriptor` type can provide a reference to a corresponding file descriptor (Unix int or Win32 HANDLE) for the *NormInstance*. For a given `NormInstanceHandle`, the `NormGetDescriptor()` function can be used to retrieve a `NormDescriptor` value that may, in turn, used in appropriate system calls (e.g. `select()` or `MsgWaitForMultipleObjects()`) to asynchronously monitor the NORM protocol engine for notification events (see `NormEvent` description).

4.1.13. NormFlushMode

The `NormFlushMode` type consists of the following enumeration:

```
enum NormFlushMode
{
    NORM_FLUSH_NONE,
    NORM_FLUSH_PASSIVE,
    NORM_FLUSH_ACTIVE
};
```

The use and interpretation of these values is given in the descriptions of `NormStreamFlush()` and `NormStreamSetAutoFlush()` functions.

4.1.14. NormProbingMode

The `NormProbingMode` type consists of the following enumeration:

```
enum NormProbingMode
{
    NORM_PROBE_NONE,
    NORM_PROBE_PASSIVE,
    NORM_PROBE_ACTIVE
};
```

The use and interpretation of these values is given in the description of `NormSetGrttProbingMode()` function.

4.1.15. NormSyncPolicy

The `NormSyncPolicy` type consists of the following enumeration:

```
enum NormSyncPolicy
{
    NORM_SYNC_CURRENT,
    NORM_SYNC_ALL
};
```

The use and interpretation of these values is given in the descriptions of the `NormSetDefaultSyncPolicy()` function.

4.1.16. NormNackingMode

The `NormNackingMode` type consists of the following enumeration:

```
enum NormNackingMode
{
    NORM_NACK_NONE,
    NORM_NACK_INFO_ONLY,
    NORM_NACK_NORMAL
};
```

The use and interpretation of these values is given in the descriptions of the `NormSetDefaultNackingMode()`, `NormNodeSetNackingMode()` and `NormObjectSetNackingMode()` functions.

4.1.17. NormRepairBoundary

The `NormRepairBoundary` types consists of the following enumeration:

```
enum NormRepairBoundary
{
    NORM_BOUNDARY_BLOCK,
    NORM_BOUNDARY_OBJECT
};
```

The interpretation of these values is given in the descriptions of the `NormSetDefaultRepairBoundary()` and `NormNodeSetRepairBoundary()` functions.

4.1.18. NormAckingStatus

The `NormAckingStatus` consist of the following enumeration:

```
enum NormAckingStatus
{
    NORM_ACK_INVALID,
    NORM_ACK_FAILURE,
    NORM_ACK_PENDING,
    NORM_ACK_SUCCESS
};
```

The interpretation of these values is given in the descriptions of the `NormGetAckingStatus()` function.

4.2. API Initialization and Operation

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.

4.2.1. NormCreateInstance()

4.2.1.1. Synopsis

```
#include <normApi.h>

NormInstanceHandle NormCreateInstance(bool priorityBoost = false);
```

4.2.1.2. 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 `NormCreateSession()`, etc. The optional *`priorityBoost`* parameter, when set to a value of true, specifies that the NORM protocol engine thread be run with higher priority scheduling. On Win32 platforms, this corresponds to `THREAD_PRIORITY_TIME_CRITICAL` and on Unix systems with the `sched_setscheduler()` API, an attempt to get the maximum allowed `SCHED_FIFO` priority is made. The use of this option should be carefully evaluated since, depending upon the application's scheduling priority and NORM API usage, this may have adverse effects instead of a guaranteed performance increase!

4.2.1.3. 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.

4.2.2. NormDestroyInstance()

4.2.2.1. Synopsis

```
#include <normApi.h>

void NormDestroyInstance(NormInstanceHandle instanceHandle);
```

4.2.2.2. Description

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

4.2.2.3. Return Values

The function has no return value.

4.2.3. NormStopInstance()

4.2.3.1. Synopsis

```
#include <normApi.h>

void NormStopInstance(NormInstanceHandle instanceHandle);
```

4.2.3.2. Description

This function immediately stops the NORM protocol engine thread corresponding to the given *`instanceHandle`* parameter. It also posts a "dummy" notification event so that if another thread is blocked on a call to `NormGetNextEvent()`, that thread will be released. Hence, for some multi-threaded uses of the NORM API, this function may be useful as a preliminary step to safely coordinate thread shutdown before a call is made to `NormDestroyInstance()`. After `NormStopInstance()` is called and any pending events posted prior to its call have been retrieved, `NormGetNextEvent()` will return a value of `false`.

When this function is invoked, state for any *NormSessions* associated with the given instance is "frozen". The complementary function, `NormRestartInstance()` can be subsequently used to "unfreeze" and resume NORM protocol operation (a new thread is created and started).

4.2.3.3. Return Values

The function has no return value.

4.2.4. NormRestartInstance()

4.2.4.1. Synopsis

```
#include <normApi.h>

bool NormRestartInstance(NormInstanceHandle instanceHandle);
```

4.2.4.2. Description

This function creates and starts an operating system thread to resume NORM protocol engine operation for the given *`instanceHandle`* that was previously stopped by a call to `NormStopInstance()`. It is not expected that this function will be used often, but there may be special application cases where "freezing" and later resuming NORM protocol operation may be useful.

4.2.4.3. Return Values

The function returns `true` when the NORM protocol engine thread is successfully restarted, and `false` otherwise.

4.2.5. NormSetCacheDirectory()

4.2.5.1. Synopsis

```
#include <normApi.h>

bool NormSetCacheDirectory(NormInstanceHandle    instanceHandle,
                           const char*           cachePath);
```

4.2.5.2. Description

This function sets the directory path used by receivers to cache newly-received `NORM_OBJECT_FILE` content. The *`instanceHandle`* parameter specifies the NORM protocol engine instance (all *NormSessions* associated with that *`instanceHandle`* share the same cache path) and the *`cachePath`* is a string specifying a valid (and writable) directory path.

4.2.5.3. Return Values

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.

4.2.6. NormGetNextEvent()

4.2.6.1. Synopsis

```
#include <normApi.h>

bool NormGetNextEvent(NormInstanceHandle instanceHandle,
                      NormEvent*         theEvent);
```

4.2.6.2. Description

This function retrieves the next available NORM protocol event from the protocol engine. The *`instanceHandle`* 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. For expected reliable protocol operation, the application should make every attempt to retrieve and process NORM notification events in a timely manner.

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 (int for Unix or HANDLE for WIN32) suitable for asynchronous input/output (I/O) notification using such system calls the Unix `select()` or Win32 `WaitForMultipleObjects()` calls. The descriptor is signaled when a notification event is pending and a call to `NormGetNextEvent()` will not block.`NormGetNextEvent`

4.2.6.3. Return Values

The function returns `true` when a NormEvent is successfully retrieved, and `false` otherwise. Note that a return value of `false` does *not* indicate an error or signify end of NORM operation.

4.2.6.4. NORM Notification Event Types

The following table enumerates the possible `NormEvent` values and describes how these notifications should be interpreted as they are retrieved by the application via the `NormGetNextEvent()` function call.

Sender Notifications:	
`NORM_TX_QUEUE_VACANCY`	This event indicates that there is room for additional transmit objects to be enqueued, or, if the handle of `NORM_OBJECT_STREAM` is given in the corresponding event "object" field, the application may successfully write to the indicated stream object. Note this event is not dispatched until a call to `NormFileEnqueue()`, `NormDataEnqueue()`, or `NormStreamWrite()` fails because of a filled transmit cache or stream buffer.
`NORM_TX_QUEUE_EMPTY`	This event indicates the NORM protocol engine has no new data pending transmission and the application may enqueue additional objects for transmission. If the handle of a sender `NORM_OBJECT_STREAM` is given in the corresponding event "object" field, this indicates the stream transmit buffer has been emptied and the sender application may write to the stream (Use of `NORM_TX_QUEUE_VACANCY` may be preferred for this purpose since it allows the application to keep the NORM protocol engine busier sending data, resulting in higher throughput when attempting very high transfer rates).
`NORM_TX_FLUSH_COMPLETED`	This event indicates that the flushing process the NORM sender observes when it no longer has data ready for transmission has completed. The completion of the flushing process is a reasonable indicator (with a sufficient NORM "robust factor" value) that the receiver set no longer has any pending repair requests. Note the use of NORM's optional positive acknowledgement feature is more deterministic in this regards, but this notification is useful when there are non-acking (NACK-

	only) receivers. The default NORM robust factor of 20 (20 flush messages are sent at end-of-transmission) provides a high assurance of reliable transmission, even with packet loss rates of 50%.
`NORM_TX_WATERMARK_COMPLETED`	This event indicates that the flushing process initiated by a prior application call to `NormSetWatermark()` has completed The posting of this event indicates the appropriate time for the application to make a call `NormGetAckingStatus()` to determine the results of the watermark flushing process.
`NORM_TX_OBJECT_SENT`	This event indicates that the transport object referenced by the event's "object" field has completed at least one pass of total transmission. Note that this does not guarantee that reliable transmission has yet completed; only that the entire object content has been transmitted. Depending upon network behavior, several rounds of NACKing and repair transmissions may be required to complete reliable transfer.
`NORM_TX_OBJECT_PURGED`	This event indicates that the NORM protocol engine will no longer refer to the transport object identified by the event's "object' field. Typically, this will occur when the application has enqueued more objects than space available within the set sender transmit cache bounds (see `NormSetTxCacheBounds()`). Posting of this notification means the application is free to free any resources (memory, files, etc) associated with the indicated "object". After this event, the given "object" handle (`NormObjectHandle`) is no longer valid unless it is specifically retained by the application.`NormObjectHandle`
`NORM_TX_CMD_SENT`	This event indicates that an application-defined command previously enqueued with a call to `NormSendCommand()` has been transmitted, including any repetition.
`NORM_TX_RATE_CHANGED`	This event indicates that NORM Congestion Control operation has adjusted the transmission rate. The `NormGetTxRate()` call may be used to retrieve the new corresponding transmission rate. Note that if `NormSetCongestionControl()` was called with its *`adjustRate`* parameter set to `false`, then no actual rate change has occurred and the rate value returned by `NormGetTxRate()` reflects a "suggested" rate and not the actual transmission rate.
`NORM_LOCAL_SENDER_CLOSED`	This event is posted when the NORM protocol engine completes the "graceful shutdown" of its participation as a sender in the indicated "session" (see `NormStopSender()`).
`NORM_CC_ACTIVE`	This event indicates that congestion control feedback from receivers has begun to be received (This also implies that receivers in the group are actually present and can be used as a cue to begin data transmission.). Note that congestion control must be enabled (see `NormSetCongestionControl()`) for this event to be posted. Congestion control feedback can be assumed to be received until a `NORM_CC_INACTIVE` event is posted.
`NORM_CC_INACTIVE`	This event indicates there has been no recent congestion control feedback received from the receiver set and that the local NORM sender has reached its minimum transmit rate. Applications may wish to refrain from new data transmission until a `NORM_CC_ACTIVE` event is posted. This notification is only posted when congestion control operation is enabled (see `NormSetCongestionControl()`) and a previous `NORM_CC_ACTIVE` event has occurred.
Receiver Notifications:	
`NORM_REMOTE_SENDER_NEW`	This event is posted when a receiver first receives messages from a specific remote NORM sender. This marks the beginning of the interval during which the application may reference the provided "node" handle (`NormNodeHandle`).
`NORM_REMOTE_SENDER_ACTIVE`	This event is posted when a previously inactive (or new) remote sender is detected operating as an active sender within the session.
`NORM_REMOTE_SENDER_INACTIVE`	This event is posted after a significant period of inactivity (no sender messages received) of a specific NORM sender within the session. The NORM protocol engine frees buffering resources allocated for this sender when it becomes inactive.

`NORM_REMOTE_SENDER_PURGED`	This event is posted when the NORM protocol engine frees resources for, and thus invalidates the indicated "node" handle.
`NORM_RX_OBJECT_NEW`	This event is posted when reception of a new transport object begins and marks the beginning of the interval during which the specified "object" (`NormObjectHandle`) is valid.
`NORM_RX_OBJECT_INFO`	This notification is posted when the `NORM_INFO` content for the indicated "object" is received.
`NORM_RX_OBJECT_UPDATED`	This event indicates that the identified receive "object" has newly received data content.
`NORM_RX_OBJECT_COMPLETED`	This event is posted when a receive object is completely received, including available `NORM_INFO` content. Unless the application specifically retains the "object" handle, the indicated `NormObjectHandle` becomes invalid and must no longer be referenced.
`NORM_RX_OBJECT_ABORTED`	This notification is posted when a pending receive object's transmission is aborted by the remote sender. Unless the application specifically retains the "object" handle, the indicated `NormObjectHandle` becomes invalid and must no longer be referenced.
`NORM_RX_CMD_NEW`	This event indicates that an application-defined command has been received from a remote sender. The NormEvent node element indicates the NormNodeHandle value associated with the given sender. The `NormNodeGetCommand()` call can be used to retrieve the received command content.
Miscellaneous Notifications:	
`NORM_GRTT_UPDATED`	This notification indicates that either the local sender estimate of GRTT has changed, or that a remote sender's estimate of GRTT has changed. The "sender" member of the `NormEvent` is set to `NORM_NODE_INVALID` if the local sender's GRTT estimate has changed or to the `NormNodeHandle` of the remote sender that has updated its estimate of GRTT.
`NORM_EVENT_INVALID`	This `NormEventType` indicates an invalid or "null" notification which should be ignored.

4.2.6.5. 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 exceptional cases when the function may immediately return even when no event is pending. In these cases, the return value is `false` indicating the `NormEvent` should be ignored.

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 `NORM_RX_OBJECT_UPDATED` events if not needed by the application)).

4.2.7. NormGetDescriptor()

4.2.7.1. Synopsis

```
#include <normApi.h>

NormDescriptor NormGetDescriptor(NormInstanceHandle instance);
```

4.2.7.2. Description

This function is used to retrieve a `NormDescriptor` (Unix int file descriptor or Win32 HANDLE) suitable for asynchronous I/O notification to avoid blocking calls to `NormGetNextEvent()`. A `NormDescriptor` is available for each protocol engine instance created using NormCreateInstance(). The descriptor returned 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 Unix `select()` or Win32 `WaitForMultipleObjects()` system calls can be used to detect when the `NormDescriptor` is signaled. Note that for Unix `select()` call usage, the NORM descriptor should be treated as a "read" descriptor.

4.2.7.3. Return Values

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

4.3. Session Creation and Control Functions

Whether participating in a NORM protocol session as a sender, receiver, or both, there are some common API calls used to instantiate a *NormSession* and set some common session parameters. Functions are provided to control network socket and multicast parameters. Additionally, a "user data" value may be associated with a `NormSessionHandle` for programming convenience when dealing with multiple sessions.

4.3.1. NormCreateSession()

4.3.1.1. Synopsis

```
#include <normApi.h
NormSessionHandle NormCreateSession(NormInstanceHandle       instance,
                                    const char*              address,
                                    unsigned short           port,
                                    NormNodeId               localId);
```

4.3.1.2. Description

This function creates a NORM protocol session (*NormSession*) using the *`address`* (multicast or unicast) and *`port`* 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:

`instance`	This must be a valid `NormInstanceHandle` previously obtained with a call to `NormCreateInstance()`.
`address`	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 address (along with the port 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 address parameter value is thus inconsequential for this special case).
`port`	This must be a valid, unused port number corresponding to the desired NORM session address. See the address parameter description for more details.
`localId`	The *`localId`* parameter specifies the `NormNodeId` that should be used to identify the application's presence in the *NormSession*. All participant's in a *NormSession* should use unique *`localId`* values. The application may specify a value of `NORM_NODE_ANY` or `NORM_NODE_ANY` for the *`localId`* 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 `NormNodeId` 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.

4.3.1.3. 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.

4.3.2. NormDestroySession()

4.3.2.1. Synopsis

```
#include <normApi.h>

void NormDestroySession(NormSessionHandle sessionHandle);
```

4.3.2.2. Description

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

4.3.2.3. Return Values

This function has no returned values.

4.3.3. NormSetUserData()

4.3.3.1. Synopsis

```
#include <normApi.h>

void NormSetUserData(NormSessionHandle sessionHandle,
                     const void*       userData);
```

4.3.3.2. Description

This function allows the application to attach a value to the previously-created *NormSession* instance specified by the *`sessionHandle`* parameter. This value is not used or interpreted by NORM, but is available to the application for use at the programmer's discretion. The set *`userData`* value can be later retrieved using the `NormGetUserData()` function call.

4.3.3.3. Return Values

This function has no returned values.

4.3.4. NormGetUserData()

4.3.4.1. Synopsis

```
#include <normApi.h>

const void* NormGetUserData(NormSessionHandle sessionHandle);
```

4.3.4.2. Description

This function retrieves the "user data" value set for the specified *`sessionHandle`* with a prior call to `NormSetUserData()`.

4.3.4.3. Return Values

This function returns the user data value set for the specified session. If no user data value has been previously set, a NULL (i.e., `(const void*)0`) value is returned.

4.3.5. NormGetLocalNodeId()

4.3.5.1. Synopsis

```
#include <normApi.h>

NormNodeId NormGetLocalNodeId(NormSessionHandle sessionHandle);
```

4.3.5.2. Description

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

4.3.5.3. Return Valuess

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

4.3.6. NormSetTxPort()

4.3.6.1. Synopsis

```
#include <normApi.h>

bool NormSetTxPort(NormSessionHandle sessionHandle,
                   unsigned short    txPort,
                   bool              enableReuse = false,
                   const char*       txBindAddress = (const char*)0);
```

4.3.6.2. Description

This function is used to force NORM to use a specific port number for UDP packets sent for the specified *`sessionHandle`*. Additionally, it can optionally enable reuse of the specified port number and/or specify a specific source address binding that is used for packet transmission. By default, NORM uses separate port numbers for packet transmission and session packet reception (the receive port is specified as part of the `NormCreateSession()` call), allowing the operating system to pick a freely available port for transmission. This call allows the application to pick a specific port number for transmission, and furthermore allows the application to even specify the same port number for transmission as is used for reception. However, the use of separate transmit/receive ports allows NORM to discriminate when unicast feedback is occurring and thus it is not generally recommended that the transmit port be set to the same value as the session receive port.

The *`enableReuse`* parameter, when set to `true`, allows that the specified port may be reused for multiple sessions, but care must be taken when enabling this option. The *`txBindAddress`* parameter allows specification of a specific source address binding for packet transmission. The specified address MUST be a valid unicast IP address assigned and configured for the host system. Additionally, the address specified must be compatible with multicast routing and/or the interfaces specified in any calls to `NormSetMulticastInterface()` for the given session when IP multicast is used.

When the *`txPort`* is set equal to the session port number and a *`txBindAddress`* is not specified or set equal to the session address, a single socket is used for both transmission and reception. If the same port number is desired for both packet transmission and reception, *and* a specific source address binding is set, then the *`enableReuse`* parameter MUST be (and is automatically) set to `true` for successful operation. In this case, the receive socket is bound to session address if it is multicast and the transmit socket is bound to the specified *`txAddress`* although both are bound to the same port number.

Note this call MUST be made *before* any calls to `NormStartSender()` or `NormStartReceiver()` for the given session to work as described.

4.3.6.3. Return Values

This function returns `true` upon success and `false` upon failure. Failure will occur if a *`txBindAddress`* is provided that does not correspond to a valid, configured IP address for the local host system.

4.3.7. NormSetTxOnly()

4.3.7.1. Synopsis

```
#include <normApi.h>

void NormSetTxOnly(NormSessionHandle sessionHandle,
                   bool              txOnly,
                   bool              connectToSessionAddress = false);
```

4.3.7.2. Description

This function limits the *NormSession* to perform NORM sender functions only. It also limits the underlying NORM UDP socket usage to open only a single transmit socket (tx_socket) and does *not* open or bind a receive socket for the given session address or port number. Thus, if this property is set, any NORM receivers MUST enable unicast feedback via a call to the NormSetDefaultUnicastNack() or appropriate NormNodeSetUnicastNack() function in order for their feedback messages (NACKs and ACKs) to be received by this sender. The purpose of this function is to allow NORM sender sessions to be created as separate process from a corresponding NORM receiver session for the same session address and port number. By default (when this call is not made), a *NormSession*, even when acting as only a sender (see NormStartSender()) opens two separate UDP sockets including a "receive" socket bound to the session port number and a "transmit" socket used for message transmission and reception of unicast feedback messages when receivers are so configured.

The optional *`connectToSessionAddress`* parameter, when set to `true`, causes the underlying NORM code to "`connect()`" the UDP socket to the session (remote receiver) address and port number. If the corresponding NORM remote receiver instance uses NormSetTxPort() to set its transmit port to the same as the session port number, the result is a unique binding between this "tx only" sender instance and the remote NORM receiver instance. With proper use of NormSetRxPortReuse(), this allows multiple senders to be properly associated (i.e., binded with respect to UDP socket packet demultiplexing) with multiple receivers on a single host (all using the same session port number). Note the NORM receiver MUST also use the NormSetDefaultUnicastNack() call so that its feedback messages are directed to the "tx only" sender address/port. The motivation for this API call is to allow systems have NORM sender and receiver instances in separate processes supporting a set (e.g. a mesh) of unicast connections to other hosts. The only constraint is that the senders uses a "tx port" number that is different from the "rx port" number. This enables firewall configurations that only open a pair of UDP ports and allow for connection among an arbitrary number of hosts. This option is really only relevant for unicast NORM sessions.

4.3.7.3. Return Values

This function has no return values.

4.3.8. NormSetRxPortReuse()

4.3.8.1. Synopsis

```
#include <normApi.h>

void NormSetRxPortReuse(NormSessionHandle session,
                        bool              enableReuse,
                        const char*       rxBindAddress = (const char*)0,
                        const char*       senderAddress = (const char*)0,
                        UINT16            senderPort = 0);
```

4.3.8.2. Description

This function allows the user to control the port reuse and binding behavior for the receive socket used for the given NORM *`sessionHandle`*. When the *`enablReuse`* parameter is set to `true`, reuse of the *NormSession* port number by multiple NORM instances or sessions is enabled.

If the optional *`rxBindAddress`* is supplied (an IP address or host name in string form), the socket will bind() to the given address when it is opened in a call to NormStartReceiver() or NormStartSender(). The *`rxBindAddress`* MUST be the session multicast address (if it is a multicast session) or a valid local unicast address in the case of NORM unicast operation. This binding limits the socket to receive only packets destined for the specified *`rxBindAddress`*. This allows multiple NORM sessions to reuse the same port number, but use different multicast addresses (or allow for multiple NORM sessions for multiple local unicast addresses).

The optional *`senderAddress`* and *`senderPort`* parameters can be used to connect() the underlying NORM receive socket to specific address/port. This limits the socket to receiving only packets from the specified *`senderAddress`*/*`senderPort`*. This, with receive port reuse enabled, allows for multiple NORM receiver instances to be listening to different NORM senders and have proper UDP socket demultiplexing occur. Note that it is also possible to have single NORM receiver receive transmissions from multiple senders, but in some cases it may be desirable for separate NORM processes or threads to be used to handle reception from separate senders. Thus, this socket binding option is provided.

When this call is not made in any form, the default socket binding to IP address INADDR_ANY (equivalent to when this call is made and *`rxBindAddress`* is set to `NULL`) allows the *NormSession* receive socket to receive any multicast or unicast transmissions to the session port number provided in the call to `NormCreateSession()`. This allows a NORM receiver to receive from senders sending to a multicast session address or the receiver's unicast address. As mentioned, enabling port reuse and binding the session destination address allows multiple NORM sessions on the same port number, but participating in different multicast groups.

Note this call MUST be made *before* any calls to `NormStartSender()` or `NormStartReceiver()` for the given *`sessionHandle`* to succeed.

This call could also be used in conjunction with `NormSetMulticastInterface()` so that multiple *NormSessions*, using the same port and multicast address, could separately cover multiple network interfaces (and some sort of application-layer bridging of reliable multicast could be realized if desired).

4.3.8.3. Return Values

This function has no return values.

4.3.9. NormSetMulticastInterface()

4.3.9.1. Synopsis

```
#include <normApi.h>

bool NormSetMulticastInterface(NormSessionHandle session,
                               const char*       interfaceName);
```

4.3.9.2. Description

This function specifies which host network interface is used for IP Multicast transmissions and group membership. This should be called *before* any call to `NormStartSender()` or `NormStartReceiver()` is made so that the IP multicast group is joined on the proper host interface. 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`".

4.3.9.3. Return Values

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

4.3.10. NormSetSSM()

4.3.10.1. Synopsis

```
#include <normApi.h>

bool NormSetSSM(NormSessionHandle session,
                const char*       sourceAddress);
```

4.3.10.2. Description

This function sets the source address for Source-Specific Multicast (SSM) operation. This should be called *before* any call to `NormStartSender()` or `NormStartReceiver()` is made so that the proper group join is done. The receiver application MUST also use the `NormSetDefaultUnicastNack()` call so that feedback traffic is directed back to appropriate sender.

4.3.10.3. Return Values

A return value of `true` indicates success while a return value of `false` indicates that the specified source address was invalid. Note that if a valid IP address is specified but is improper for SSM (e.g., an IP multicast address) the later calls to `NormStartSender()` or `NormStartReceiver()` may fail.

4.3.11. NormSetTTL()

4.3.11.1. Synopsis

```
#include <normApi.h>

bool NormSetTTL(NormSessionHandle session,
                unsigned char     ttl);
```

4.3.11.2. Description

This function specifies the time-to-live (*`ttl`*) for IP Multicast datagrams generated by NORM for the specified *`sessionHandle`*. The IP TTL field limits the number of router "hops" that a generated multicast packet may traverse before being dropped. For example, if TTL is equal to one, the transmissions will be limited to the local area network (LAN) of the host computers network interface. Larger TTL values should be specified to span large networks. Also note that some multicast router configurations use artificial "TTL threshold" values to constrain some multicast traffic to an administrative boundary. In these cases, the NORM TTL setting must also exceed the router "TTL threshold" in order for the NORM traffic to be allowed to exit the administrative area.

4.3.11.3. Return Values

A return value of `true` indicates success while a return value of `false` indicates that the specified *`ttl`* could not be set. This function will always return `true` if made before calls to `NormStartSender()` or `NormStartReceiver()`. However, those calls may fail if the desired *`ttl`* value cannot be set.

4.3.12. NormSetTOS()

4.3.12.1. Synopsis

```
#include <normApi.h>

bool NormSetTOS(NormSessionHandle       sessionHandle,
                unsigned char           tos);
```

4.3.12.2. Description

This function specifies the type-of-service (*`tos`*) field value used in IP Multicast datagrams generated by NORM for the specified *`sessionHandle`*. The IP TOS field value can be used as an indicator that a "flow" of packets may merit special Quality-of-Service (QoS) treatment by network devices. Users should refer to applicable QoS information for their network to determine the expected interpretation and treatment (if any) of packets with explicit TOS marking.

4.3.12.3. Return Values

A return value of `true` indicates success while a return value of `false` indicates that the specified *`tos`* could not be set. This function will always return `true` if made before calls to `NormStartSender()` or `NormStartReceiver()`. However, those calls may fail if the desired *`tos`* value cannot be set.

4.3.13. NormSetLoopback()

4.3.13.1. Synopsis

```
#include <normApi.h>

void NormSetLoopback(NormSessionHandle sessionHandle,
                     bool              loopbackEnable);
```

4.3.13.2. Description

This function enables or disables loopback operation for the indicated NORM *`sessionHandle`*. If *`loopbackEnable`* is set to `true`, loopback operation is enabled which allows the application to receive its own message traffic. Thus, an application which is both actively receiving and sending may receive its own transmissions. Note it is expected that this option would be principally be used for test purposes and that applications would generally not need to transfer data to themselves. If *`loopbackEnable`* is false, the application is prevented from receiving its own NORM message transmissions. By default, loopback operation is disabled when a *NormSession* is created.

4.3.13.3. Return Values

This function has no return values.

4.3.14. NormSetFragmentation()

4.3.14.1. Synopsis

```
#include <normApi.h>

bool NormSetFragmentation(NormSessionHandle sessionHandle,
                          bool              fragmentation);
```

4.3.14.2. Description

This function sets an underlying socket option that enables or disables IP datagram fragmentation by network intermediate systems according to whether the *`fragmentation`* parameter is set to a value of `true` or `false`, respectively. If set to `true` to enable fragmentation, the DF (don't fragment) bit of the headers of NORM UDP/IP packets sent will be cleared. Otherwise the DF bit is set and packets will not be fragmented by network devices if they exceed a link Maximum Transmission Unit (MTU) and will instead be dropped. For IP Multicast destinations, some operating systems may always set the DF bit of transmitted packets, regardless of the setting here and the underlying socket option status. Typically, the DF bit is set (i.e., fragmentation disabled) by default on most operating systems.

This call is not currently functional on the Mac OSX system that does not support the needed `IP_MTU_DISCOVER` or `IP_DONTFRAG` socket options.

4.3.14.3. Return Values

This function returns `true` upon success and `false` upon failure.

4.4. NORM Sender Functions

The functions described in this section apply only to NORM sender operation. Applications may participate strictly as senders or as receivers, or may act as both in the context of a NORM protocol session. The NORM sender is responsible for most parameters pertaining to its transmission of data. This includes transmission rate, data segmentation sizes, FEC coding parameters, stream buffer sizes, etc.

4.4.1. NormStartSender()

4.4.1.1. Synopsis

```
#include <normApi.h>

bool NormStartSender(NormSessionHandle        sessionHandle,
                     NormSessionId            instanceId,
                     unsigned long            bufferSpace,
                     unsigned short           segmentSize,
                     unsigned char            blockSize,
                     unsigned char            numParity);
```

4.4.1.2. 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 `NormEvent` notifications. The parameters required for this function call include:

`sessionHandle`	This must be a valid `NormSessionHandle` previously obtained with a call to `NormCreateSession()`.
`instanceId`	Application-defined value used as the `instance_id` field of NORM sender messages for the application's participation within a session. Receivers can detect when a sender has terminated and restarted if the application uses different *`instanceId`* values when initiating sender operation. For example, a robust application could cache previous *`instanceId`* values in non-volatile storage and gracefully recover (without confusing receivers) from a total system shutdown and reboot by using a new *`instanceId`* value upon restart.
`bufferSpace`	This specifies the maximum memory space (in bytes) 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 + *`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 "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 (*`blockSize`* + *`numParity`*) <= `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 "`n - 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` - *`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 "*`intance_id`*" 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.

4.4.1.3. 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 sessionHandle)` function to retrieve a more specific error indication for this and other functions.*)

4.4.2. NormStopSender()

4.4.2.1. Synopsis

```
#include <normApi.h>

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

4.4.2.2. Description

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

4.4.2.3. Return Values

This function has no return values.

4.4.3. NormSetTxRate()

4.4.3.1. Synopsis

```
#include <normApi.h>

void NormSetTxRate(NormSessionHandle sessionHandle,
                   double            rate);
```

4.4.3.2. Description

This function sets the transmission *`rate`* (in bits per second (bps)) limit used for *NormSender* transmissions for the given *`sessionHandle`*. 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 are sent. For `NORM_OBJECT_STREAM` transmissions, this is the maximum rate allowed for transmission (i.e. if the application writes to the stream at a lower rate, a lower average NORM transmission rate will occur). 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, if enabled, may quickly readjust the transmission rate.

4.4.3.3. Return Values

This function has no return values.

4.4.4. NormGetTxRate()

4.4.4.1. Synopsis

```
#include <normApi.h>

double NormGetTxRate(NormSessionHandle sessionHandle);
```

4.4.4.2. Description

This function retrieves the current sender transmission rate in units of bits per second (bps) for the given *`sessionHandle`*. When NORM congestion control is enabled (see `NormSetCongestionControl()`), this reflects the current rate set (or suggested) by NORM congestion control operation. Otherwise, this returns the rate that was set with the `NormSetTxRate()` call.

4.4.4.3. Return Values

This function returns the sender transmission rate in units of bits per second (bps).

4.4.5. NormSetTxSocketBuffer()

4.4.5.1. Synopsis

```
#include <normApi.h>

bool NormSetTxSocketBuffer(NormSessionHandle sessionHandle,
                           unsigned int      bufferSize);
```

4.4.5.2. Description

This function can be used to set a non-default socket buffer size for the UDP socket used by the specified NORM *`sessionHandle`* for data transmission. The *`bufferSize`* parameter specifies the desired socket buffer size in bytes. Large transmit socket buffer sizes may be necessary to achieve high transmission rates when NORM, as a user-space process, is unable to precisely time its packet transmissions. Similarly, NORM receivers may need to set large receive socket buffer sizes to achieve successful, sustained high data rate reception (see `NormSetRxSocketBuffer()`). Typically, it is more important to set the receive socket buffer size (see `NormSetRxSocketBuffer()`) as this maintains reliability (i.e. by avoiding receive socket buffer overflow) at high data rates while setting a larger transmit socket buffer size allows higher average transmission rates to be achieved.

4.4.5.3. Return Values

This function returns `true` upon success and `false` upon failure. Possible failure modes include an invalid *`sessionHandle`* parameter, a call to `NormStartReceiver()` or `NormStartSender()` has not yet been made for the session, or an invalid *`bufferSize`* was given. Note some operating systems may require additional system configuration to use non-standard socket buffer sizes.

4.4.6. NormSetFlowControl()

4.4.6.1. Synopsis

```
#include <normApi.h>

void NormSetFlowControl(NormSessionHandle sessionHandle,
                        double            flowControlFactor);
```

4.4.6.2. Description

This function controls a scaling factor that is used for sender timer-based flow control for the the specified NORM *`sessionHandle`*. Timer-based flow control works by preventing the NORM sender application from enqueueing new transmit objects or stream data that would purge "old" objects or stream data when there has been recent NACK activity for those old objects or data. If the *`flowControlFactor`* is set to `ZERO`, then the flow control mechanism is effectively disabled. Larger *`flowControlFactor`* values enforce more robust flow control by forcing the sender to maintain state longer, but then larger transmit buffer, stream buffer, transmit cache bounds and receive cache limits (see `NormStartSender()`, `NormStreamOpen()`, `NormSetTxCacheBounds()`, and `NormSetRxCacheLimit()`, respectively) may be needed to maintain throughput in larger <delay*bandwidth, loss> conditions. Effectively, a larger *`flowControlFactor`* can favor reliability over throughput when buffer-constrained.

The *`flowControlFactor`* is used to compute a delay time for when a sender buffered object (or block of stream data) may be released (i.e. purged) after transmission or applicable NACKs reception. The delay time function is:

```
flowControlDelay = flowControlFactor * GRTT * (backoffFactor + 1)
```

where the "`GRTT`" is the sender's advertised GRTT estimate and the `backoffFactor` is the sender's configured timer-based feedback scaling factor.

The default value (when this function is not called) of the *`flowControlFactor`* is `2.0`. Note that a NORM application can also implement more explicit, deterministic flow control through use of the `NormSetWatermark()` API call, potentially even requiring positive acknowledgement of older data before enqueueing new data. Note that using the `NormSetWatermark()` API call with a `NORM_NODE_NONE` member in acking node list to force a "full" watermark flush is somewhat equivalent to timer-based flow control with a *`flowControlFactor`* equal to `2.0 * txRobustFactor`.

If such explicit flow control is implemented by the application, then a reduced *`flowControlFactor`* (or even `ZERO`) may be used. If "push mode" is enabled for a `NORM_OBJECT_STREAM` (see `NormStreamSetPushEnable()`), then flow control has no effect for the stream.

4.4.6.3. Return Values

This function has no return values.

4.4.7. NormSetCongestionControl()

4.4.7.1. Synopsis

```
#include <normApi.h>

void NormSetCongestionControl(NormSessionHandle sessionHandle,
                              bool              enable,
                              bool              adjustRate = true);
```

4.4.7.2. Description

This function enables (or disables) the NORM sender congestion control operation for the session designated by the *`sessionHandle`* parameter. For best operation, this function should be called before the call to `NormStartSender()` is made, but congestion control operation can be dynamically enabled/disabled during the course of sender operation. If the value of the *`enable`* parameter is `true`, congestion control operation is enabled while it is disabled for enable equal to `false`. When congestion control operation is enabled, the NORM sender automatically adjusts its transmission rate based on feedback from receivers. If bounds on transmission rate have been set (see `NormSetTxRateBounds()`) the rate adjustment will remain within the set bounds. The application will be notified of any changes to the sender transmission rate via a `NormEvent` of type `NORM_TX_RATE_CHANGED`.

The rate set by `NormSetTxRate()` has no effect when congestion control operation is enabled, *unless* the *`adjustRate`* parameter here is set to `false`. When the *`adjustRate`* parameter is set to `false`, the NORM Congestion Control operates as usual, with feedback collected from the receiver set and the "current limiting receiver" identified, except that no actual adjustment is made to the sender's transmission rate. I.e., the transmission rate that was set by `NormSetTxRate()` is observed by the sender regardless of the feedback received. The `NORM_TX_RATE_CHANGED` notification will still occur as if the rate were being adjusted and the value returned by `NormGetTxRate()` reflects the rate that would have been used had the *`adjustRate`* parameter been enabled even though no actual rate change has occurred. The purpose of this variation of NORM Congestion Control operation is to allow applications to get a "suggested" rate from the NORM-CC mechanism. But, it is important to note that this "suggested" rate may or may not be appropriate since the operation of the NORM-CC algorithm is somewhat dependent on the associated NORM sender load on the network. For example, the "suggested" rate may be artificially high if the sender application has not been correspondingly setting the rate and actively transmitting data at that rate. This optional mode of operation is provided for EXPERIMENTAL purposes and is NOT RECOMMENDED for typical use of NORM.

NORM's congestion algorithm provides rate adjustment to fairly compete for available network bandwidth with other TCP, NORM, or similarly governed traffic flows.

(*TBD - Describe the `NormSetEcnSupport()` function as this experimental option matures.*)

4.4.7.3. Return Values

This function has no return values.

4.4.8. NormSetTxRateBounds()

4.4.8.1. Synopsis

```
#include <normApi.h>

bool NormSetTxRateBounds(NormSessionHandle sessionHandle,
                               double            rateMin,
                               double            rateMax);
```

4.4.8.2. Description

This function sets the range of sender transmission rates within which the NORM congestion control algorithm is allowed to operate for the given *`sessionHandle`*. 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, the call "`NormSetTxRateBounds(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 be adjusted to stay within the set bounds.

4.4.8.3. Return Values

This function returns `true` upon success. If both *`rateMin`* and *`rateMax`* are greater than or equal to zero, but (*`rateMax`* < *`rateMin`*), the rate bounds will remain unset or unchanged and the function will return `false`.

4.4.9. NormSetTxCacheBounds()

4.4.9.1. Synopsis

```
#include <normApi.h>

void NormSetTxCacheBounds(NormSessionHandle sessionHandle,
                                NormSize          sizeMax,
                                unsigned int      countMin,
                                unsigned int      countMax);
```

4.4.9.2. Description

This function sets limits that define the number and total size of pending transmit objects a NORM sender will allow to be enqueued by the application. Setting these bounds to large values means the NORM protocol engine will keep history and state for previously transmitted objects for a larger interval of time (depending upon the transmission rate) when the application is actively enqueueing additional objects in response to `NORM_TX_QUEUE_EMPTY` notifications. This can allow more time for receivers suffering degraded network conditions to make repair requests before the sender "purges" older objects from its "transmit cache" when new objects are enqueued. A `NORM_TX_OBJECT_PURGED` notification is issued when the enqueuing of a new transmit object causes the NORM transmit cache to overflow, indicating the NORM sender no longer needs to reference the designated old transmit object and the application is free to release related resources as needed.

The *`sizeMax`* parameter sets the maximum total size, in bytes, of enqueued objects allowed, providing the constraints of the *`countMin`* and *`countMax`* parameters are met. The *`countMin`* parameter sets the minimum number of objects the application may enqueue, regardless of the objects' sizes and the *`sizeMax`* value. For example, the default *`sizeMax`* value is 20 Mbyte and the default *`countMin`* is 8, thus allowing the application to always have at least 8 pending objects enqueued for transmission if it desires, even if their total size is greater than 20 Mbyte. Similarly, the *`countMax`* parameter sets a ceiling on how many objects may be enqueued, regardless of their total sizes with respect to the *`sizeMax`* setting. For example, the default *`countMax`* value is 256, which means the application is never allowed to have more than 256 objects pending transmission enqueued, even if they are 256 very small objects. Note that *`countMax`* must be greater than or equal to *`countMin`* and *`countMin`* is recommended to be at least two.

Note that in the case of `NORM_OBJECT_FILE` objects, some operating systems impose limits (e.g. 256) on how many open files a process may have at one time and it may be appropriate to limit the *`countMax`* value accordingly. In other cases, a large *`countMin`* or *`countMax`* may be desired to allow the NORM sender to act as virtual cache of files or other data available for reliable transmission. Future iterations of the NRL NORM implementation may support alternative NORM receiver "group join" policies that would allow the receivers to request transmission of cached content.

The utility of the `NormRequeueObject()` API call also depends on the parameters set by this function. The `NormRequeueObject()` call will only succeed when the given *`objectHandle`* corresponds to an object maintained in the NORM senders "transmit cache".

4.4.9.3. Return Values

This function has no return value.

4.4.10. NormSetAutoParity()

4.4.10.1. Synopsis

```
#include <normApi.h>

void NormSetAutoParity(NormSessionHandle sessionHandle,
                       unsigned char     autoParity);
```

4.4.10.2. Description

This function sets the quantity of proactive "auto parity" `NORM_DATA` messages sent at the end of each FEC coding block. By default (i.e., *`autoParity`* `= 0`), FEC content is sent only in response to repair requests (NACKs) from receivers. But, by setting a non-zero value for *`autoParity`*, the sender can automatically accompany each coding block of transport object source data segments ((`NORM_DATA` messages) with the set number of FEC segments. The number of source symbol messages (segments) per FEC coding block is determined by the *`blockSize`* parameter used when `NormStartSender()` was called for the given *`sessionHandle`*.

The use of proactively-sent "auto parity" may eliminate the need for any receiver NACKing to achieve reliable transfer in networks with low packet loss. However, note that the quantity of "auto parity" set adds overhead to transport object transmission. In networks with a predictable level of packet loss and potentially large round-trip times, the use of "auto parity" may allow lower latency in the reliable delivery process. Also, its use may contribute to a smaller amount of receiver feedback as only receivers with exceptional packet loss may need to NACK for additional repair content.

The value of *`autoParity`* set must be less than or equal to the *`numParity`* parameter set when `NormStartSender()` was called for the given *`sessionHandle`*.

4.4.10.3. Return Values

This function has no return values.

4.4.11. NormGetGrttEstimate()

4.4.11.1. Synopsis

```
#include <normApi.h>

double NormGetGrttEstimate(NormSessionHandle sessionHandle);
```

4.4.11.2. Description

This function returns the sender's current estimate(in seconds) of group round-trip timing (GRTT) for the given NORM session. This function may be useful for applications to leverage for other purposes the assessment of round-trip timing made by the NORM protocol engine. For example, an application may scale its own timeouts based on connectivity delays among participants in a NORM session. Note that the `NORM_GRTT_UPDATED` event is posted (see `NormGetNextEvent()`) by the NORM protocol engine to indicate when changes in the local sender or remote senders' GRTT estimate occurs.

4.4.11.3. Return Values

This function returns the current sender group round-trip timing (GRTT) estimate (in units of seconds). A value of `-1.0` is returned if an invalid session value is provided.

4.4.12. NormSetGrttEstimate()

4.4.12.1. Synopsis

```
#include <normApi.h>

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

4.4.12.2. Description

This function sets the sender's estimate of group round-trip time (GRTT) (in units of seconds) for the given NORM *`sessionHandle`*. 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 `NormSetGrttProbingMode()`). The *`grtt`* value (in seconds) 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.

Also note that the advertised GRTT estimate can also be limited by transmission rate. When the sender transmission rate is low, the GRTT is also governed to a lower bound of the nominal packet transmission interval (i.e., `1/txRate`). This maintains the "event driven" nature of the NORM protocol with respect to receiver reception of NORM sender data and commands.

4.4.12.3. Return Values

This function has no return values.

4.4.13. NormSetGrttMax()

4.4.13.1. Synopsis

```
#include <normApi.h>

void NormSetGrttMax(NormSessionHandle sessionHandle,
                    double            grttMax);
```

4.4.13.2. Description

This function sets the sender's maximum advertised GRTT value for the given NORM *`sessionHandle`*. The *`grttMax`* parameter, in units of seconds, limits the GRTT used by the group for scaling protocol timers, regardless of larger measured round trip times. The default maximum for the NRL NORM library is 10 seconds. See the `NormSetGrttEstimate()` function description for the purpose of the NORM GRTT measurement process.

4.4.13.3. Return Values

This function has no return values.

4.4.14. NormSetGrttProbingMode()

4.4.14.1. Synopsis

```
#include <normApi.h>

void NormSetGrttProbingMode(NormSessionHandle sessionHandle,
                            NormProbingMode   probingMode);
```

4.4.14.2. Description

This function sets the sender's mode of probing for round trip timing measurement responses from the receiver set for the given NORM *`sessionHandle`*. Possible values for the *`probingMode`* parameter include `NORM_PROBE_NONE`, `NORM_PROBE_PASSIVE`, and `NORM_PROBE_ACTIVE`. The default probing mode is `NORM_PROBE_ACTIVE`. In this mode, the receiver set explicitly acknowledges NORM sender GRTT probes (`(NORM_CMD(CC)` messages) with `NORM_ACK` responses that are group-wise suppressed. Note that NORM receivers also will include their response to GRTT probing piggy-backed on any `NORM_NACK` messages sent in this mode as well to minimize feedback.

Note that the `NORM_PROBE_ACTIVE` probing mode is required and automatically set when NORM congestion control operation is enabled (see `NormSetCongestionControl()`). Thus, when congestion control is enabled, the `NormSetGrttProbingMode()` function has no effect.

If congestion control operation is not enabled, the NORM application may elect to reduce the volume of feedback traffic by setting the *`probingMode`* to `NORM_PROBE_PASSIVE`. Here, the NORM sender still transmits `NORM_CMD(CC)` probe messages multiplexed with its data transmission, but the receiver set does not explicitly acknowledge these probes. Instead the receiver set is limited to opportunistically piggy-backing responses when `NORM_NACK` messages are generated. Note that this may, in some cases, introduce some opportunity for bursts of large volume receiver feedback when the sender's estimate of GRTT is incorrect due to the reduced probing feedback. But, in some controlled network environments, this option for passive probing may provide some benefits in reducing protocol overhead.

Finally, the *`probingMode`* can be set to `NORM_PROBE_NONE` to eliminate the overhead (and benefits) of NORM GRTT measurement entirely. In this case, the sender application must explicitly set its estimate of GRTT using the `NormSetGrttEstimate()` function. See this function for a description of the purpose of the NORM GRTT measurement.

4.4.14.3. Return Values

This function has no return values.

4.4.15. NormSetGrttProbingInterval()

4.4.15.1. Synopsis

```
#include <normApi.h>

void NormSetGrttProbingInterval(NormSessionHandle sessionHandle,
                                double            intervalMin,
                                double            intervalMax);
```

4.4.15.2. Description

This function controls the sender GRTT measurement and estimation process for the given NORM *`sessionHandle`*. The NORM sender multiplexes periodic transmission of `NORM_CMD`(CC) messages with its ongoing data transmission or when data transmission is idle. When NORM congestion control operation is enabled, these probes are sent once per RTT of the current limiting receiver (with respect to congestion control rate). In this case the *`intervalMin`* and *`intervalMax`* parameters (in units of seconds) control the rate at which the sender's estimate of GRTT is updated. At session start, the estimate is updated at *`intervalMin`* and the update interval time is doubled until *`intervalMax`* is reached. This dynamic allows for a rapid initial estimation of GRTT and a slower, steady-state update of GRTT. When congestion control is disabled and NORM GRTT probing is enabled (`(NORM_PROBE_ACTIVE` or `NORM_PROBE_PASSIVE`) the *`intervalMin`* and *`intervalMax`* values also determine the rate at which `NORM_CMD`(CC) probes are transmitted by the sender. Thus by setting larger values for *`intervalMin`* and *`intervalMax`*, the NORM sender application can reduce the overhead of the GRTT measurement process. However, this also reduces the ability of NORM to adapt to changes in GRTT.

The default NORM GRTT *`intervalMin`* and *`intervalMax`* values, i.e., when this call is not made, are `1.0` second and `30.0` seconds, respectively.

4.4.15.3. Return Values

This function has no return values.

4.4.16. NormSetBackoffFactor()

4.4.16.1. Synopsis

```
#include <normApi.h>

void NormSetBackoffFactor(NormSessionHandle sessionHandle,
                          double            backoffFactor);
```

4.4.16.2. Description

This function sets the sender's "backoff factor" for the given *`sessionHandle`*. The *`backoffFactor`* (in units of seconds) is used to scale various timeouts related to the NACK repair process. The sender advertises its *`backoffFactor`* setting to the receiver group in NORM protocol message headers. The default *`backoffFactor`* for NORM sessions is `4.0` seconds. The *`backoffFactor`* is used to determine the maximum time that receivers may delay NACK transmissions (and other feedback messages) as part of NORM's probabilistic feedback suppression technique. For example, the maximum NACK delay time is *`backoffFactor`*`*GRTT`. Thus a large `backoffFactor` value introduces latency into the NORM repair process. However, a small backoffFactor value causes feedback suppression to be less effective and increases the risk of feedback implosion for large receiver group sizes.

The default setting of `4.0` provides reasonable feedback suppression for moderate to large group sizes when multicast feedback is possible. The NORM specification recommends a `backoffFactor` value of `6.0` when unicast feedback is used. However, for demanding applications (with respect to repair latency) when group sizes are modest, a small (even `0.0`) `backoffFactor` value can be specified to reduce the latency of reliable data delivery.

4.4.16.3. Return Values

This function has no return values.

4.4.17. NormSetGroupSize()

4.4.17.1. Synopsis

```
#include <normApi.h>

void NormSetGroupSize(NormSessionHandle sessionHandle,
                      unsigned int      groupSize);
```

4.4.17.2. Description

This function sets the sender's estimate of receiver group size for the given *`sessionHandle`*. The sender advertises its *`groupSize`* setting to the receiver group in NORM protocol message headers that, in turn, use this information to shape the distribution curve of their random timeouts for the timer-based, probabilistic feedback suppression technique used in the NORM protocol. Note that the *`groupSize`* estimate does not have to be very accurate and values within an order of magnitude of the actual group size tend to produce acceptable performance.

The default *`groupSize`* setting in NORM is `1,000` and thus can work well for a wide range of actual receiver group sizes. The penalty of an overly large estimate is statistically a little more latency in reliable data delivery with respect to the round trip time and some potential for excess feedback. A substantial underestimation of *`groupSize`* increases the risk of feedback implosion. Currently, the NORM implementation does not attempt to automatically measure *`groupSize`* from receiver feedback. Applications could add their own mechanism for this (perhaps keeping explicit track of group membership), or it is possible that future versions of the NRL NORM implementation may have some provision for automatic *`groupSize`* estimation by the sender based on receiver feedback messages.

4.4.17.3. Return Values

This function has no return values.

4.4.18. NormSetTxRobustFactor()

4.4.18.1. Synopsis

```
#include <normApi.h>

void NormSetTxRobustFactor(NormSessionHandle sessionHandle,
                           int               txRobustFactor);
```

4.4.18.2. Description

This routine sets the "robustness factor" used for various NORM sender functions. These functions include the number of repetitions of "robustly-transmitted" NORM sender commands such as `NORM_CMD(FLUSH)` or similar application-defined commands, and the number of attempts that are made to collect positive acknowledgement from receivers. These commands are distinct from the NORM reliable data transmission process, but play a role in overall NORM protocol operation. The default *`txRobustFactor`* value is `20`. This relatively large value makes the NORM sender end-of-transmission flushing and positive acknowledgement collection functions somewhat immune from packet loss. However, for some applications, the default value may make the NORM protocol more "chatty" than desired (particularly if flushing is invoked often). In other situations where the network connectivity may be intermittent or extremely lossy, it may be useful to actually increase this value. The default value (`20`) is expected to provide reasonable operation across a wide range of network conditions and application types. Since this value is not communicated among NORM participants as part of the protocol operation, it is important that applications consistently set this value among all applications participating in a NORM session.

Setting *`txRobustFactor`* to a value of `-1` makes the redundant transmission of these commands continue indefinitely until completion. For example, with positive acknowledgement collection, the request process will continue indefinitely until all recipients requested acknowledge or the request is canceled by the application. Similarly, flushing commands would be transmitted repeatedly until data transmission is resumed. Typically, setting *`txRobustFactor`* to `-1` is not recommended.

4.4.18.3. Return Values

This function has no return values.

4.4.19. NormFileEnqueue()

4.4.19.1. Synopsis

```
#include <normApi.h>

NormObjectHandle NormFileEnqueue(NormSessionHandle      sessionHandle,
                                 const char*            filename,
                                 const char*            infoPtr = NULL,
                                 unsigned int           infoLen = 0);
```

4.4.19.2. Description

This function enqueues a file for transmission within the specified NORM *`sessionHandle`*. Note that `NormStartSender()` must have been previously called before files or any transport objects may be enqueued and transmitted. The *`fileName`* parameter specifies the path to the file to be transmitted. The NORM protocol engine read and writes directly from/to file system storage for file transport, potentially providing for a very large virtual "repair window" as needed for some applications. While relative paths with respect to the "current working directory" may be used, it is recommended that full paths be used when possible. The optional *`infoPtr`* and *`infoLen`* parameters are used to associate `NORM_INFO` content with the sent transport object. The maximum allowed *`infoLen`* corresponds to the *`segmentSize`* used in the prior call to `NormStartSender()`. The use and interpretation of the `NORM_INFO` content is left to the application's discretion. Example usage of `NORM_INFO` content for `NORM_OBJECT_FILE` might include file name, creation date, MIME-type or other information which will enable NORM receivers to properly handle the file when reception is complete.

The application is allowed to enqueue multiple transmit objects within in the "transmit cache" bounds (see `NormSetTxCacheBounds()`) and enqueued objects are transmitted (and repaired as needed) within the limits determined by automated congestion control (see `NormSetCongestionControl()`) or fixed rate (see `NormSetTxRate()`) parameters.

4.4.19.3. Return Values

A `NormObjectHandle` is returned which the application may use in other NORM API calls as needed. This handle can be considered valid until the application explicitly cancels the object's transmission (see `NormObjectCancel()`) or a `NORM_TX_OBJECT_PURGED` event is received for the given object. Note the application may use the `NormObjectRetain()` method if it wishes to refer to the object after the `NORM_TX_OBJECT_PURGED` notification. In this case, the application, when finished with the object, must use `NormObjectRelease()` to free any resources used or else a memory leak condition will result. A value of `NORM_OBJECT_INVALID` is return upon error. Possible failure conditions include the specified session is not operating as a NormSender, insufficient memory resources were available, or the "transmit cache" limits have been reached and all previously enqueued NORM transmit objects are pending transmission. Also the call will fail if the *`infoLen`* parameter exceeds the local *NormSender* *`segmentSize`* limit.

4.4.20. NormDataEnqueue()

4.4.20.1. Synopsis

```
#include <normApi.h>

NormObjectHandle NormDataEnqueue(NormSessionHandle      sessionHandle,
                                 const char*            dataPtr,
                                 unsigned int           dataLen,
                                 const char*            infoPtr = NULL,
                                 unsigned int           infoLen = 0);
```

4.4.20.2. Description

This function enqueues a segment of application memory space for transmission within the specified NORM *`sessionHandle`*. Note that `NormStartSender()` MUST have been previously called before files or any transport objects may be enqueued and transmitted. The *`dataPtr`* parameter must be a valid pointer to the area of application memory to be transmitted and the *`dataLen`* parameter indicates the quantity of data to transmit. The NORM protocol engine read and writes directly from/to application memory space so it is important that the application does not modify (or deallocate) the memory space during the time the NORM protocol engine may access this area. After calling `NormDataEnqueue()` for a specific application "dataPtr" memory space, the application MUST NOT deallocate (or change the contents of) that memory space until a `NORM_TX_OBJECT_PURGED` notification is received for the given object or the application itself explicitly cancels the object's transmission (see `NormObjectCancel()`).

The optional *`infoPtr`* and *`infoLen`* parameters are used to associate `NORM_INFO` content with the sent transport object. The maximum allowed *`infoLen`* corresponds to the *`segmentSize`* used in the prior call to `NormStartSender()`. The use and interpretation of the `NORM_INFO` content is left to the application's discretion. Example usage of `NORM_INFO` content for `NORM_OBJECT_DATA` might include application-defined data typing or other information which will enable NORM receiver applications to properly interpret the received data when reception is complete. Of course, it is possible that the application may embed such typing information in the object data content itself. This is left to the application's discretion.

The application is allowed to enqueue multiple transmit objects within in the "transmit cache" bounds (see `NormSetTxCacheBounds()`) and enqueued objects are transmitted (and repaired as needed) within the limits determined by automated congestion control (see `NormSetCongestionControl()`) or fixed rate (see `NormSetTxRate()`) parameters.

4.4.20.3. Return Values

A `NormObjectHandle` is returned which the application may use in other NORM API calls as needed. This handle can be considered valid until the application explicitly cancels the object's transmission (see `NormObjectCancel()`) or a `NORM_TX_OBJECT_PURGED` event is received for the given object. Note the application may use the `NormObjectRetain()` method if it wishes to refer to the object after the `NORM_TX_OBJECT_PURGED` notification. In this case, the application, when finished with the object, must use `NormObjectRelease()` to free any resources used or else a memory leak condition will result. A value of `NORM_OBJECT_INVALID` is return upon error. Possible failure conditions include the specified session is not operating as a NormSender, insufficient memory resources were available, or the "transmit cache" limits have been reached and all previously enqueued NORM transmit objects are pending transmission. Also the call will fail if the *`infoLen`* parameter exceeds the local *NormSender* *`segmentSize`* limit.

4.4.21. NormRequeueObject()

4.4.21.1. Synopsis

```
#include <normApi.h>

boolNormRequeueObject(NormSessionHandle sessionHandle,
                      NormObjectHandle  objectHandle);
```

4.4.21.2. Description

This function allows the application to resend (or reset transmission of) a `NORM_OBJECT_FILE` or `NORM_OBJECT_DATA` transmit object that was previously enqueued for the indicated *`sessionHandle`*. This function is useful for applications sending to silent (non-NACKing) receivers as it enables the receivers to take advantage of multiple retransmissions of objects (including any auto-parity set, see `NormSetAutoParity()`) to more robustly receive content. The *`objectHandle`* parameter must be a valid transmit `NormObjectHandle` that has not yet been "purged" from the sender's transmit queue. Upon success, the specified object will be fully retransmitted using the same NORM object transport identifier as was used on its initial transmission. This call may be made at any time to restart transmission of a previously-enqueued object, but the `NORM_TX_OBJECT_SENT` or `NORM_TX_FLUSH_COMPLETED` notifications can serve as good cues for an appropriate time to resend an object. If multiple objects are re-queued, they will be resent in order of their initial enqueueing.

The transmit cache bounds set by `NormSetTxCacheBounds()` determine the number of previously-sent objects retained in the sender's transmit queue and that are thus eligible to be requeued for retransmission. An object may be requeued via this call multiple times, but each distinct requeue should be done after an indication such as `NORM_TX_OBJECT_SENT` or `NORM_TX_FLUSH_COMPLETED` for the given object. Otherwise, the object will simply be reset from its current transmission point to transmit from the beginning (i.e. restart). Note that the object type `NORM_OBJECT_STREAM` cannot currently be requeued.

(TBD - should a "numRepeats" parameter be added to this function?)

4.4.21.3. Return Values

A value of `true` is returned upon success and a value of `false` is returned upon failure. Possible reasons for failure include an invalid *`objectHandle`* was provided (i.e. a non-transmit object or transmit object that has been "purged" from the transmit queue (see `NORM_TX_OBJECT_PURGED`)) or the provided object was of type `NORM_OBJECT_STREAM`.

4.4.22. NormStreamOpen()

4.4.22.1. Synopsis

```
#include <normApi.h>

NormObjectHandle NormStreamOpen(NormSessionHandle      sessionHandle,
                                unsigned int           bufferSize,
                                const char*            infoPtr = NULL,
                                unsigned int           infoLen = 0);
```

4.4.22.2. Description

This function opens a `NORM_OBJECT_STREAM` sender object and enqueues it for transmission within the indicated *`sessionHandle`*. NORM streams provide reliable, in-order delivery of data content written to the stream by the sender application. Note that no data is sent until subsequent calls to `NormStreamWrite()` are made unless `NORM_INFO` content is specified for the stream with the *`infoPtr`* and *`infoLen`* parameters. Example usage of `NORM_INFO` content for `NORM_OBJECT_STREAM` might include application-defined data typing or other information which will enable NORM receiver applications to properly interpret the received stream as it is being received. The NORM protocol engine buffers data written to the stream for original transmission and repair transmissions as needed to achieve reliable transfer. The *`bufferSize`* parameter controls the size of the stream's "repair window" which limits how far back the sender will "rewind" to satisfy receiver repair requests.

NORM, as a NACK-oriented protocol, currently lacks a mechanism for receivers to explicitly feedback flow control status to the sender unless the sender application specifically leverages NORM's optional positive-acknowledgement (ACK) features. Thus, the *`bufferSize`* selection plays an important role in reliable delivery of NORM stream content. Generally, a larger *`bufferSize`* value is safer with respect to reliability, but some applications may wish to limit how far the sender rewinds to repair receivers with poor connectivity with respect to the group at large. Such applications may set a smaller *`bufferSize`* to avoid the potential for large latency in data delivery (i.e. favor peak delivery latency over full reliability). This may result in breaks in the reliable delivery of stream data to some receivers, but this form of quasi-reliability while limiting latency may be useful for some types of applications (e.g. reliable real-time messaging, video or sensor or media data transport). Note that NORM receivers can quickly, automatically "resync" to the sender after such breaks if the application leverages the application message boundary recovery features of NORM (see `NormStreamMarkEom()`).

Note that the current implementation of NORM is designed to support only one active stream per session, and that any `NORM_OBJECT_DATA` or `NORM_OBJECT_FILE` objects enqueued for transmission will not begin transmission until an active stream is closed. Applications requiring multiple streams or concurrent file/data transfer SHOULD generally instantiate multiple *NormSessions* as needed.

Note there is no corresponding "open" call for receiver streams. Receiver `NORM_OBJECT_STREAM`s are automatically opened by the NORM protocol engine and the receiver applications is notified of new streams via the `NORM_RX_OBJECT_NEW` notification (see `NormGetNextEvent()`).

4.4.22.3. Return Values

A `NormObjectHandle` is returned which the application may use in other NORM API calls as needed. This handle can be considered valid until the application explicitly cancels the object's transmission (see `NormObjectCancel()`) or a `NORM_TX_OBJECT_PURGED` event is received for the given object. Note the application may use the `NormObjectRetain()` method if it wishes to refer to the object after the `NORM_TX_OBJECT_PURGED` notification. In this case, the application, when finished with the object, must use `NormObjectRelease()` to free any resources used or else a memory leak condition will result. A value of `NORM_OBJECT_INVALID` is return upon error. Possible failure conditions include the specified session is not operating as a *NormSender*, insufficient memory resources were available, or the "transmit cache" bounds have been reached and all previously enqueued NORM transmit objects are pending transmission. Also the call will fail if the *`infoLen`* parameter exceeds the local *NormSender* *`segmentSize`* limit.

4.4.23. NormStreamClose()

4.4.23.1. Synopsis

```
#include <normApi.h>

void NormStreamClose(NormObjectHandle streamHandle,
                     bool             graceful = false);
```

4.4.23.2. Description

This function halts transfer of the stream specified by the *`streamHandle`* parameter and releases any resources used unless the associated object has been explicitly retained by a call to `NormObjectRetain()`. No further calls to `NormStreamWrite()` will be successful for the given *`streamHandle`*. The optional graceful parameter, when set to a value of true, may be used by NORM senders to initiate "graceful" shutdown of a transmit stream. In this case, the sender application will be notified that stream has (most likely) completed reliable transfer via the `NORM_TX_OBJECT_PURGED` notification upon completion of the graceful shutdown process. When the *`graceful`* option is set to `true`, receivers are notified of the stream end via an "stream end" stream control code in `NORM_DATA` message and will receive a `NORM_RX_OBJECT_COMPLETED` notification after all received stream content has been read. Otherwise, the stream is immediately terminated, regardless of receiver state. In this case, this function is equivalent to the `NormObjectCancel()` routine and may be used for sender or receiver streams. So, it is expected this function (`NormStreamClose()`) will typically be used for transmit streams by NORM senders.

4.4.23.3. Return Values

This function has no return values.

4.4.24. NormStreamWrite()

4.4.24.1. Synopsis

```
#include <normApi.h>

unsigned int NormStreamWrite(NormObjectHandle streamHandle
                             const char*      buffer,
                             unsigned int     numBytes);
```

4.4.24.2. Description

This function enqueues data for transmission within the NORM stream specified by the *`streamHandle`* parameter. The *`buffer`* parameter must be a pointer to the data to be enqueued and the *`numBytes`* parameter indicates the length of the data content. Note this call does not block and will return immediately. The return value indicates the number of bytes copied from the provided buffer to the internal stream transmission buffers. Calls to this function will be successful unless the stream's transmit buffer space is fully occupied with data pending original or repair transmission if the stream's "push mode" is set to false (default, see `NormStreamSetPushEnable()` for details). If the stream's "push mode" is set to true, a call to `NormStreamWrite()` will always result in copying of application data to the stream at the cost of previously enqueued data pending transmission (original or repair) being dropped by the NORM protocol engine. While NORM NACK-based reliability does not provide explicit flow control, there is some degree of implicit flow control in limiting writing new data to the stream against pending repairs. Other flow control strategies are possible using the `NormSetWatermark()` function.`NormSetWatermark()` function.

The `NormEvent` values `NORM_TX_QUEUE_EMPTY` and `NORM_TX_QUEUE_VACANCY` are posted with the `NormEvent`*`::object`* field set to a valid sender stream `NormObjectHandle` to indicate when the stream is ready for writing via this function. Note that the `NORM_TX_QUEUE_VACANCY` event type is posted only after the stream's transmit buffer has been completely filled. Thus, the application must make a call to `NormStreamWrite()` that copies less than the requested *`numBytes`* value (return value less than *`numBytes`*) before additional `NORM_TX_QUEUE_VACANCY` events are posted for the given *`streamHandle`* (i.e., the event type is not re-posted until the application has again filled the available stream transmit buffer space). By cueing off of `NORM_TX_QUEUE_EMPTY`, the application can write its "freshest" available data to the stream, but by cueing off of `NORM_TX_QUEUE_VACANCY`, an application can keep the NORM protocol engine busiest, to achieve the maximum possible throughput at high data rates.

4.4.24.3. Return Values

This function returns the number of bytes of data successfully enqueued for NORM stream transmission. If the underlying send stream buffer is full, this function may return zero or a value less than the requested *`numBytes`*.

4.4.25. NormStreamFlush()

4.4.25.1. Synopsis

```
#include <normApi.h>

void NormStreamFlush(NormObjectHandle streamHandle,
                     bool             eom = false,
                     NormFlushMode    flushMode = NORM_FLUSH_PASSIVE);
```

4.4.25.2. Description

This function causes an immediate "flush" of the transmit stream specified by the *`streamHandle`* parameter. Normally, unless `NormStreamSetAutoFlush()` has been invoked, the NORM protocol engine buffers data written to a stream until it has accumulated a sufficient quantity to generate a `NORM_DATA` message with a full payload (as designated by the *`segmentSize`* parameter of the `NormStartSender()` call). This results in most efficient operation with respect to protocol overhead. However, for some NORM streams, the application may not wish wait for such accumulation when critical data has been written to a stream. The default stream "flush" operation invoked via `NormStreamFlush()` for *`flushMode`* equal to `NORM_FLUSH_PASSIVE` causes NORM to immediately transmit all enqueued data for the stream (subject to session transmit rate limits), even if this results in `NORM_DATA` messages with "small" payloads. If the optional *`flushMode`* parameter is set to `NORM_FLUSH_ACTIVE`, the application can achieve reliable delivery of stream content up to the current write position in an even more proactive fashion. In this case, the sender additionally, actively transmits `NORM_CMD`(FLUSH) messages after any enqueued stream content has been sent. This immediately prompt receivers for repair requests which reduces latency of reliable delivery, but at a cost of some additional messaging. Note any such "active" flush activity will be terminated upon the next subsequent write to the stream. If *`flushMode`* is set to `NORM_FLUSH_NONE`, this call has no effect other than the optional end-of-message marking described here.

The optional *`eom`* parameter, when set to `true`, allows the sender application to mark an end-of-message indication (see `NormStreamMarkEom()`) for the stream and initiate flushing in a single function call. The end-of-message indication causes NORM to embed the appropriate message start byte offset in the `NORM_DATA` message generated following a subsequent write to the stream with the `NORM_FLAGS_MSG_START` flag. This mechanism provide a means for automatic application message boundary recovery when receivers join or re-sync to a sender mid-stream.

Note that frequent flushing, particularly for `NORM_FLUSH_ACTIVE` operation, may result in more NORM protocol activity than usual, so care must be taken in application design and deployment when scalability to large group sizes is expected.

4.4.25.3. Return Values

This function has no return values.

4.4.26. NormStreamSetAutoFlush()

4.4.26.1. Synopsis

```
#include <normApi.h>

void NormStreamSetAutoFlush(NormObjectHandle streamHandle
                            NormFlushMode    flushMode);
```

4.4.26.2. Description

This function sets "automated flushing" for the NORM transmit stream indicated by the *`streamHandle`* parameter. By default, a NORM transmit stream is "flushed" only when explicitly requested by the application (see `NormStreamFlush()`). However, to simplify programming, the NORM API allows that automated flushing be enabled such that the "flush" operation occurs every time the full requested buffer provided to a `NormStreamWrite()` call is successfully enqueued. This may be appropriate for messaging applications where the provided buffers corresponds to an application messages requiring immediate, full transmission. This may make the NORM protocol perhaps more "chatty" than its typical "bulk transfer" form of operation, but can provide a useful capability for some applications.

Possible values for the *`flushMode`* parameter include `NORM_FLUSH_NONE`, `NORM_FLUSH_PASSIVE`, and `NORM_FLUSH_ACTIVE`. The default setting for a NORM stream is `NORM_FLUSH_NONE` where no flushing occurs unless explicitly requested via `NormStreamFlush()`. By setting the automated *`flushMode`* to `NORM_FLUSH_PASSIVE`, the only action taken is to immediately transmit any data that has been written to the stream, even if "runt" `NORM_DATA` messages (with payloads less than the *NormSender `segmentSize`* parameter) are generated as a result. If `NORM_FLUSH_ACTIVE` is specified, the automated flushing operation is further augmented with the additional transmission of `NORM_CMD(FLUSH)` messages to proactively excite the receiver group for repair requests.

4.4.26.3. Return Values

This function has no return values.

4.4.27. NormStreamSetPushEnable()

4.4.27.1. Synopsis

```
#include <normApi.h>

void NormStreamSetPushEnable(NormObjectHandle streamHandle,
                             bool             pushEnable);
```

4.4.27.2. Description

This function controls how the NORM API behaves when the application attempts to enqueue new stream data for transmission when the associated stream's transmit buffer is fully occupied with data pending original or repair transmission. By default (*`pushEnable`* = `false`), a call to `NormStreamWrite()` will return a zero value under this condition, indicating it was unable to enqueue the new data. However, if *`pushEnable`* is set to `true` for a given *`streamHandle`*, the NORM protocol engine will discard the oldest buffered stream data (even if it is pending repair transmission or has never been transmitted) as needed to enqueue the new data. Thus a call to `NormStreamWrite()` will never fail to copy data. This behavior may be desirable for applications where it is more important to quickly delivery new data than to reliably deliver older data written to a stream. The default behavior for a newly opened stream corresponds to *`pushEnable`* equals `false`. This limits the rate to which an application can write new data to the stream to the current transmission rate and status of the reliable repair process.

4.4.27.3. Return Values

This function has no return values.

4.4.28. NormStreamHasVacancy()

4.4.28.1. Synopsis

```
#include <normApi.h>

bool NormStreamHasVacancy(NormObjectHandle streamHandle);
```

4.4.28.2. Description

This function can be used to query whether the transmit stream, specified by the *`streamHandle`* parameter, has buffer space available so that the application may successfully make a call to `NormStreamWrite()`. Normally, a call to `NormStreamWrite()` itself can be used to make this determination, but this function can be useful when "push mode" has been enabled (see the description of the `NormStreamSetPushEnable()` function) and the application wants to avoid overwriting data previously written to the stream that has not yet been transmitted. Note that when "push mode" is enabled, a call to `NormStreamWrite()` will always succeed, overwriting previously-enqueued data if necessary. Normally, this function will return true after a `NORM_TX_QUEUE_VACANCY` notification has been received for a given NORM stream object.

4.4.28.3. Return Values

This function returns a value of `true` when there is transmit buffer space to which the application may write and `false` otherwise.

4.4.29. NormStreamMarkEom()

4.4.29.1. Synopsis

```
#include <normApi.h>

void NormStreamMarkEom(NormObjectHandle streamHandle);
```

4.4.29.2. Description

This function allows the application to indicate to the NORM protocol engine that the last data successfully written to the stream indicated by *`streamHandle`* corresponded to the end of an application-defined message boundary. The end-of-message indication given here will cause the NORM protocol engine to embed the appropriate message start byte offset in the `NORM_DATA` message generated that contains the data for the subsequent application call to NormStreamWrite(). Use of this end-of-message marking enables NORM receivers to automatically re-sync to application-defined message boundaries when joining (or re-joining) a NORM session already in progress.

4.4.29.3. Return Values

This function has no return values.

4.4.30. NormSetWatermark()

4.4.30.1. Synopsis

```
#include <normApi.h>

bool NormSetWatermark(NormSessionHandle sessionHandle,
                      NormObjectHandle  objectHandle,
                      bool              overrideFlush = true);
```

4.4.30.2. Description

This function specifies a "watermark" transmission point at which NORM sender protocol operation should perform a flushing process and/or positive acknowledgment collection for a given *`sessionHandle`*. For `NORM_OBJECT_FILE` and `NORM_OBJECT_DATA` transmissions, the positive acknowledgement collection will begin when the specified object has been completely transmitted. The *`objectHandle`* parameter must be a valid handle to a previously-created sender object (see `NormFileEnqueue()`, `NormDataEnqueue()`, or `NormStreamOpen()`). For `NORM_OBJECT_STREAM` transmission, the positive acknowledgment collection begins immediately, using the current position (offset of most recent data written) of the sender stream as a reference.

The functions `NormAddAckingNode()` and `NormRemoveAckingNode()` are used to manage the list of `NormNodeId` values corresponding to NORM receivers that are expected to explicitly acknowledge the watermark flushing messages transmitted by the sender. Note that the `NormNodeId` `NORM_NODE_NONE` may be included in the list. Inclusion of `NORM_NODE_NONE` forces the watermark flushing process to proceed through a full `NORM_ROBUST_FACTOR` number of rounds before completing, prompting any receivers that have not completed reliable reception to the given watermark point to NACK for any repair needs. If NACKs occur, the flushing process is reset and repeated until completing with no NACKs for data through the given watermark transmission point are received. Thus, even without explicit positive acknowledgment, the sender can use this process (by adding `NORM_NODE_NONE` to the session's list of "acking nodes") for a high level of assurance that the receiver set is "happy" (completed reliable data reception) through the given object (or stream transmission point).

The event `NORM_TX_WATERMARK_COMPLETED` is posted for the given session when the flushing process or positive acknowledgment collection has completed. The process completes as soon as all listed receivers have responded unless `NORM_NODE_NONE` is included in the "acking node" list. The sender application may use the function `NormGetAckingStatus()` to determine the degree of success of the flushing process in general or for individual `NormNodeId` values.

The flushing is conducted concurrently with ongoing data transmission and does not impede the progress of reliable data transfer. Thus the sender may still enqueue *NormObjects* for transmission (or write to the existing stream) and the positive acknowledgement collection and flushing procedure will be multiplexed with the ongoing data transmission. However, the sender application may wish to defer from or limit itself in sending more data until a `NORM_TX_WATERMARK_COMPLETED` event is received for the given session. This provides a form of sender->receiver(s) flow control which does not exist in NORM's default protocol operation. If a subsequent call is made to `NormSetWatermark()` before the current acknowledgement request has completed, the pending acknowledgment request is canceled and the new one begins.

The optional *`overrideFlush`* parameter, when set to `true`, causes the watermark acknowledgment process that is established with this function call to potentially fully supersede the usual NORM end-of-transmission flushing process that occurs. If *`overrideFlush`* is set and the "watermark" transmission point corresponds to the last transmission that will result from data enqueued by the sending application, then the watermark flush completion will terminate the usual flushing process. I.e., if positive acknowledgement of watermark is received from the full "acking node list", then no further flushing is conducted. Thus, the *`overrideFlush`* parameter should only be set when the "acking node list" contains a complete list of intended recipients. This is useful for small receiver groups (or unicast operation) to reduce the "chattiness" of NORM's default end-of-transmission flush process. Note that once the watermark flush is completed and further data enqueued and transmitted, the normal default end-of-transmission behavior will be resumed unless another "watermark" is set with *`overrideFlush`* enabled. Thus, as long as new watermarks are established by successive use of this API call, this effectively "morphs" NORM into a protocol driven by positive acknowledgement behavior.

4.4.30.3. Return Values

The function returns `true` upon successful establishment of the watermark point. The function may return `false` upon failure.

4.4.31. NormCancelWatermark()

4.4.31.1. Synopsis

```
#include <normApi.h>

bool NormCancelWatermark(NormSessionHandle sessionHandle);
```

4.4.31.2. Description

This function cancels any "watermark" acknowledgement request that was previously set via the `NormSetWatermark()` function for the given *`sessionHandle`*. The status of any NORM receivers that may have acknowledged prior to cancellation can be queried using the `NormGetAckingStatus()` function even after `NormCancelWatermark()` is called. Typically, applications should wait until a event has been posted, but in some special cases it may be useful to terminate the acknowledgement collection process early.

4.4.31.3. Return Values

The function has no return values.

4.4.32. NormAddAckingNode()

4.4.32.1. Synopsis

```
#include <normApi.h>

bool NormAddAckingNode(NormSessionHandle sessionHandle,
                       NormNodeId        nodeId);
```

4.4.32.2. Description

When this function is called, the specified *`nodeId`* is added to the list of `NormNodeId` values (i.e., the "acking node" list) used when NORM sender operation performs positive acknowledgement (ACK) collection for the specified *`sessionHandle`*. The optional NORM positive acknowledgement collection occurs when a specified transmission point (see `NormSetWatermark()`) is reached or for specialized protocol actions such as positively-acknowledged application-defined commands.

Additionally the special value of *`nodeId`* equal to `NORM_NODE_NONE` may be set to force the watermark flushing process through a full `NORM_ROBUST_FACTOR` number of rounds regardless of actual acking nodes. Otherwise the flushing process is terminated when all of the nodes in the acking node list have responded. Setting a "watermark" and forcing a full flush process with the special `NORM_NODE_NONE` value of *`nodeId`* enables the resultant `NORM_TX_WATERMARK_COMPLETED` notification to be a indicator with high (but not absolute) assurance that the receiver set has completed reliable reception of content up through the "watermark" transmission point. This provides a form of scalable reliable multicast "flow control" for NACK-based operation without requiring explicit positive acknowledgement from all group members. Note that the use of the `NORM_NODE_NONE` value may be mixed with other *`nodeId`* for a mix of positive acknowledgement collection from some nodes and a measure of assurance for the group at large.

4.4.32.3. Return Values

The function returns `true` upon success and `false` upon failure. The only failure condition is that insufficient memory resources were available. If a specific *`nodeId`* is added more than once, this has no effect.

4.4.33. NormRemoveAckingNode()

4.4.33.1. Synopsis

```
#include <normApi.h>

void NormRemoveAckingNode(NormSessionHandle sessionHandle,
                          NormNodeId        nodeId);
```

4.4.33.2. Description

This function deletes the specified *`nodeId`* from the list of `NormNodeId` values used when NORM sender operation performs positive acknowledgement (ACK) collection for the specified *`sessionHandle`*. Note that if the special *`nodeId`* value "`NORM_NODE_NONE`" has been added to the list, it too must be explicitly removed to change the watermark flushing behavior if desired.

4.4.33.3. Return Values

The function has no return values.

4.4.34. NormGetNextAckingNode()

4.4.34.1. Synopsis

```
#include <normApi.h>

NormNodeId NormGetNextAckingNode(NormSessionHandle session, bool reset = false);
```

4.4.34.2. Description

This function iteratively retrieves the `NormNodeId` values in the "acking node" list maintained by a NORM sender (see `NormAddAckingNode()`) for the given *`sessionHandle`*. If the optional *`reset`* parameter is set to a value of `true`, the first `NormNodeId` value in the list is returned and subsequent calls to `NormGetNextAckingNode()`with the *`reset`* parameter set to its default `false` value will iteratively return the remaining `NormNodeId` values contained in the list. A value of `NORM_NODE_NONE` is returned when the end of the list is reached.

The "acking node" list is populated with application calls to `NormAddAckingNode()` or auto-populated if that optional behavior is set for a NormSession. Note that this API does not enable the programmer to check if the `NORM_NODE_NONE` value itself is contained in the list. The programmer should keep track of that by other means.

The following code example illustrates how to use this call to iterate through the set of stored `NormNodeId` values and get the current "acking status" for each:

```
NormNodeId nextNodeId = NormGetNextAckingNode(session, true);
while(NORM_NODE_NONE != nextNodeId)
{
    NormAckingStatus ackingStatus = NormGetAckingStatus(session, nextNodeId);
    printf("ACKing node id = %lu acking status = %d\n", nextNodeId, (int)ackingStatus);
}
```

As noted below, a good time to check the acking status of the receiver set is after a `NORM_TX_WATERMARK_COMPLETED` notification has occurred.

4.4.34.3. Return Values

The function iteratively returns `NormNodeId` values from the given session's local sender "acking node" list. A value of `NORM_NODE_NONE` is returned when the end of the list is reached.

4.4.35. NormGetAckingStatus()

4.4.35.1. Synopsis

```
#include <normApi.h>

NormAckingStatus NormGetAckingStatus(NormSessionHandle sessionHandle,
                                     NormNodeId        nodeId = NORM_NODE_ANY);
```

4.4.35.2. Description

This function queries the status of the watermark flushing process and/or positive acknowledgment collection initiated by a prior call to `NormSetWatermark()` for the given *`sessionHandle`*. In general, it is expected that applications will invoke this function after the corresponding `NORM_TX_WATERMARK_COMPLETED` event has been posted.

Setting the default parameter value *`nodeId`* = `NORM_NODE_ANY` returns a "status" indication for the overall process. Also, individual *`nodeId`* values may be queried using the `NormNodeId` values of receivers that were included in previous calls to `NormAddAckingNode()` to populate the sender session's acking node list.

If the flushing/acknowledgment process is being used for application flow control, the sender application may wish to reset the watermark and flushing process (using `NormSetWatermark()`) if the response indicates that some nodes have failed to respond. However, note that the flushing/acknowledgment process itself does elicit NACKs from receivers as needed and is interrupted and reset by any repair response that occurs. Thus, even by the time the flushing process has completed (and `NORM_TX_WATERMARK_COMPLETED` is posted) once, this is an indication that the NORM protocol has made a valiant attempt to deliver the content. Resetting the watermark process can increase robustness, but it may be in vain to repeat this process multiple times when likely network connectivity has been lost or expected receivers have failed (dropped out, shut down, etc).

4.4.35.3. Return Values

Possible return values include:

`NORM_ACK_INVALID`	The given *`sessionHandle`* is invalid or the given *`nodeId`* is not in the sender's acking list.
`NORM_ACK_FAILURE`	The positive acknowledgement collection process did not receive acknowledgment from every listed receiver (*`nodeId`* = `NORM_NODE_ANY`) or the identified *`nodeId`* did not respond.
`NORM_ACK_PENDING`	The flushing process at large has not yet completed (*`nodeId`* = `NORM_NODE_ANY`) or the given individual *`nodeId`* is still being queried for response.
`NORM_ACK_SUCCESS`	All receivers (*`nodeId`* = `NORM_NODE_ANY`) responded with positive acknowledgement or the given specific *`nodeId`* did acknowledge.

4.4.36. NormSendCommand()

4.4.36.1. Synopsis

```
#include <normApi.h>

bool NormSendCommand(NormSessionHandle session,
                     const char*       cmdBuffer,
                     unsigned int      cmdLength,
                     bool              robust = false);
```

4.4.36.2. Description

This function enqueues a NORM application-defined command for transmission. The *`cmdBuffer`* parameter points to a buffer containing the application-defined command content that will be contained in the `NORM_CMD(APPLICATION)` message payload. The *`cmdLength`* indicates the length of this content (in bytes) and MUST be less than or equal to the *`segmentLength`* value for the given *`session`* (see `NormStartSender()`). The NORM command transmission will be multiplexed with any NORM data transmission. The command is NOT delivered reliably, but can be optionally transmitted with repetition (once per GRTT) according to the NORM transmit robust factor value (see `NormSetTxRobustFactor()`) for the given session if the *`robust`* parameter is set to `true`. The command transmission is subject to any congestion control or set rate limits for the NORM session. Once the command has been transmitted (with repetition if *`robust`* is set to `true`), a `NORM_TX_CMD_SENT` notification is issued. An application can only enqueue a *single* command at a time (i.e. the `NORM_TX_CMD_SENT` notification must occur before another command can be sent). The `NormCancelCommand()` call is available to terminate command transmission if needed. Note that if a rapid succession of commands are sent it is possible that the commands may be delivered to the receivers out-of-order. Also, when repetition is requested (i.e., if *`robust`* is set to `true`) the receiver may receive duplicate copies of the same command. It is up to the application to provide any needed mechanism for detecting and/or filtering duplicate command reception.

The application-defined command feature allows NORM applications to provide some out-of-band (with respect to reliable data delivery) signaling to support session management or other functions. The reception of these "atomic" commands is relatively stateless (as compared to reliable data delivery) and thus it is possible for many senders within a group to send commands without extreme resource burden on receivers (i.e. other participants). Again, this "light-weight" signaling mechanism may be used to provide ancillary communication for the group. In the future, an additional API mechanism will be provided to support application-defined positive acknowledgement requests that could conceivably be used to help guarantee command delivery if desired.

4.4.36.3. Return Values

The function returns `true` upon success. The function may fail, returning `false`, if the session is not set for sender operation (see `NormStartSender()`), the *`cmdLength`* exceeds the configured session *`segmentLength`*, or a previously-enqueued command has not yet been sent.

4.4.37. NormCancelCommand()

4.4.37.1. Synopsis

```
#include <normApi.h>

void NormCancelCommand(NormSessionHandle session);
```

4.4.37.2. Description

This function terminates any pending `NORM_CMD(APPLICATION)` transmission that was previously initiated with the `NormSendCommand()` call. Due to the asynchrony of the NORM protocol engine thread and the application, it is possible that the command may have been already sent but the `NormCancelCommand()` call will ensure a `NORM_TX_CMD_SENT` notification is *not* issued for that prior command.

The application-defined command feature allows NORM applications to provide some out-of-band (with respect to reliable data delivery) signaling to support session management or other functions. The reception of these "atomic" commands is relatively stateless (as compared to reliable data delivery) and thus it is possible for many senders within a group to send commands without extreme resource burden on receivers (i.e. other participants). Again, this "light-weight" signaling mechanism may be used to provide ancillary communication for the group. In the future, an additional API mechanism will be provided to support application-defined positive acknowledgement requests that could conceivably be used to help guarantee command delivery if desired.

4.4.37.3. Return Values

The function has not return value.

4.5. NORM Receiver Functions

4.5.1. NormStartReceiver()

4.5.1.1. Synopsis

```
#include <normApi.h>

bool NormStartReceiver(NormSessionHandle sessionHandle,
                       unsigned long     bufferSpace);
```

4.5.1.2. Description

This function initiates the application's participation as a receiver within the *NormSession* identified by the *`sessionHandle`* parameter. The NORM protocol engine will begin providing the application with receiver-related `NormEvent` notifications, and, unless `NormSetSilentReceiver(true)` is invoked, respond to senders with appropriate protocol messages. The *`bufferSpace`* parameter is used to set a limit on the amount of *`bufferSpace`* allocated by the receiver per active *NormSender* within the session. The appropriate *`bufferSpace`* to use is a function of expected network delay*bandwidth product and packet loss characteristics. A discussion of trade-offs associated with NORM transmit and receiver buffer space selection is provided later in this document. An insufficient *`buf-`*

`ferSpace` allocation will result in potentially inefficient protocol operation, even though reliable operation may be maintained. In some cases of a large delay*bandwidth product and/or severe packet loss, a small *`bufferSpace`* allocation (coupled with the lack of explicit flow control in NORM) may result in the receiver "re-syncing" to the sender, resulting in "outages" in the reliable transmissions from a sender (this is analogous to a TCP connection timeout failure).

4.5.1.3. 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 session communication failed to open or properly configure.

4.5.2. NormStopReceiver()

4.5.2.1. Synopsis

```
#include <normApi.h>

void NormStopReceiver(NormSessionHandle sessionHandle,
                      unsigned int      gracePeriod = 0);
```

4.5.2.2. 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 `NormNodeRetain()` and `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. `"graceTime" = (`*`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 then freed.

4.5.2.3. Return Values

This function has no return values.

4.5.3. NormSetRxCacheLimit()

4.5.3.1. Synopsis

```
#include <normApi.h>

void NormSetRxCacheLimit(NormSessionHandle sessionHandle,
                         unsigned short    countMax);
```

4.5.3.2. Description

This function sets a limit on the number of outstanding (pending) *NormObjects* for which a receiver will keep state on a per-sender basis. Note that the value *`countMax`* sets a limit on the maximum consecutive range of objects that can be pending. The default value (when this function is not called) of *`countMax`* is `256`. This should be sufficient for most bulk transfer usage, but if small object sizes (e.g. small `NORM_OBJECT_DATA` messages) are being transferred, it may be useful to raise this limit in cases of high transmission speeds or large <delay*bandwidth, loss> network conditions. If the receiver cache limit is set too small (i.e. for high speed or large <delay*bandwidth> operation), the receiver may not maintain reliable reception or impact session throughput when flow control is enabled (see `NormSetFlowControl()`). The maximum allowed value of *`countMax`* is `16,384.`

If this value is changed after `NormStartReceiver()` has been called, it will only affect newly-detected remote senders, so this should typically be called before NORM receiver operation is initiated.

4.5.3.3. Return Values

This function has no return value.

4.5.4. NormSetRxSocketBuffer()

4.5.4.1. Synopsis

```
#include <normApi.h>

bool NormSetRxSocketBuffer(NormSessionHandle sessionHandle,
                           unsigned int      bufferSize);
```

4.5.4.2. Description

This function allows the application to set an alternative, non-default buffer size for the UDP socket used by the specified NORM *`sessionHandle`* for packet reception. This may be necessary for high speed NORM sessions where the UDP receive socket buffer becomes a bottleneck when the NORM protocol engine (which is running as a user-space process) doesn't get to service the receive socket quickly enough resulting in packet loss when the socket buffer overflows. The *`bufferSize`* parameter specifies the socket buffer size in bytes. Different operating systems and sometimes system configurations allow different ranges of socket buffer sizes to be set. Note that a call to `NormStartReceiver()` (or `NormStartSender()`) must have been previously made for this call to succeed (i.e., the socket must be already open).

4.5.4.3. Return Values

This function returns `true` upon success and `false` upon failure. Possible reasons for failure include, 1) the specified session is not valid, 2) that NORM "receiver" (or "sender") operation has not yet been started for the given session, or 3) an invalid *`bufferSize`* specification was given.

4.5.5. NormSetSilentReceiver()

4.5.5.1. Synopsis

```
#include <normApi.h>

void NormSetSilentReceiver(NormSessionHandle sessionHandle,
                           bool              silent,
                           INT32             maxDelay = -1);
```

4.5.5.2. Description

This function provides the option to configure a NORM receiver application as a "silent receiver". This mode of receiver operation dictates that the host does not generate any protocol messages while operating as a receiver within the specified *`sessionHandle`*. Setting the *`silent`* parameter to `true` enables silent receiver operation while setting it to `false` results in normal protocol operation where feedback is provided as needed for reliability and protocol operation. Silent receivers are dependent upon proactive FEC transmission (see `NormSetAutoParity()`) or using repair information requested by other non-silent receivers within the group to achieve reliable transfers.

The optional *`maxDelay`* parameter is most applicable for reception of the `NORM_OBJECT_STREAM` type. The default value of *`maxDelay`* `= -1` corresponds to normal operation where source data segments for incompletely-received FEC coding blocks (or transport objects) are passed to the application only when imposed buffer constraints (either the `NORM_OBJECT_STREAM` buffer size (see `NormStreamOpen()`) or the FEC receive buffer limit (see `NormStartReceiver()`) require. Thus, the default behavior (*`maxDelay`* `= -1`), causes the receiver to buffer received FEC code blocks for as long as possible (within buffer constraints as newer data arrives) before allowing the application to read the data. Hence, the receive latency (delay) can be quite long depending upon buffer size settings, transmission rate, etc. When the `maxDelay` parameter is set to a non-negative value, the value determines the maximum number of FEC coding blocks (according to a NORM sender's current transmit position) the receiver will cache an incompletely-received FEC block before giving the application the (incomplete) set of received source segments. For example, a value of *`maxDelay`* = `0` will provide the receive application with any data from the previous FEC block as soon as a subsequent FEC block is begun reception. However, this provide no protection against the possibility of out-of-order delivery of packets by the network. Therefore, if lower latency operation is desired when using silent receivers, a minimum *`maxDelay`* value of `1` is recommended. For `NORM_OBJECT_FILE` and `NORM_OBJECT_DATA`, the only impact of a non-negative *`maxDelay`* value is that previous transport objects will be immediately aborted when subsequent object begin reception. Thus, it is not usually recommended to apply a non-negative *`maxDelay`* value when `NORM_OBJECT_STREAM` is not being used.

4.5.5.3. Return Values

This function has no return values.

4.5.6. NormSetDefaultUnicastNack()

4.5.6.1. Synopsis

```
#include <normApi.h>

void NormSetDefaultUnicastNack(NormSessionHandle sessionHandle,
                               bool              enable);
```

4.5.6.2. Description

This function controls the default behavior determining the destination of receiver feedback messages generated while participating in the session. If the *`enable`* parameter is true, "unicast NACKing" is enabled for new remote senders while it is disabled for state equal to false. The NACKing behavior for current remote senders is not affected. When "unicast NACKing" is disabled (default), NACK messages are sent to the session address (usually a multicast address) and port, but when "unicast NACKing" is enabled, receiver feedback messages are sent to the unicast address (and port) based on the source address of sender messages received. For unicast NORM sessions, it is recommended that "unicast NACKing" be enabled. Note that receiver feedback messages subject to potential "unicast NACKing" include NACK-messages as well as some ACK messages such as congestion control feedback. Explicitly solicited ACK messages, such as those used to satisfy sender watermark acknowledgement requests (see `NormSetWatermark()`) are always unicast to the applicable sender. (*TBD - provide API option so that all messages are multicast.*) The default session-wide behavior for unicast NACKing can be overridden via the `NormNodeSetUnicastNack()` function for individual remote senders.

4.5.6.3. Return Values

This function has no return values.

4.5.7. NormNodeSetUnicastNack()

4.5.7.1. Synopsis

```
#include <normApi.h>

void NormNodeSetUnicastNack(NormNodeHandle senderNode,
                            bool           enable);
```

4.5.7.2. Description

This function controls the destination address of receiver feedback messages generated in response to a specific remote NORM sender corresponding to the *`senderNode`* parameter. If *`enable`* is `true`, "unicast NACKing" is enabled while it is disabled for *`enable`* equal to `false`. See the description of `NormSetDefaultUnicastNack()` for details on "unicast NACKing" behavior.

4.5.7.3. Return Values

This function has no return values.

4.5.8. NormSetDefaultSyncPolicy()

4.5.8.1. Synopsis

```
#include <normApi.h>

void NormSetDefaultSyncPolicy(NormSessionHandle sessionHandle,
                              NormSyncPolicy   syncPolicy);
```

4.5.8.2. Description

This function sets the default "synchronization policy" used when beginning (or restarting) reception of objects from a remote sender (i.e., "syncing" to the sender) for the given *`sessionHandle`*. The "synchronization policy" is the behavior observed by the receiver with regards to what objects it attempts to reliably receive (via transmissions of Negative Acknowledgements to the sender(s) or group as needed). There are currently two synchronization policy types defined:

`NORM_SYNC_CURRENT`	Attempt reception of "current" and new objects only. (default)
`NORM_SYNC_ALL`	Attempt recovery and reliable reception of all objects held in sender transmit object cache and newer objects.

The behavior of a receiver using the default `NORM_SYNC_CURRENT` policy is to attempt reliable reception only for the first received "current" and newer (with respect to the ordinal NORM object transport identifiers used by the protocol) objects from a given NORM sender. Additionally, reliable reception is only attempted when receiving a non-repair `NORM_DATA` message (or optionally a NORM positive acknowledgement request) from the *first* forward error correction (FEC) encoding block of the given object. This somewhat conservative synchronization behavior helps prevent late-joining (or otherwise "flaky" with respect to group membership) receivers from penalizing other receivers in the group by causing the sender to "rewind" and transmit older object content to satisfy the late joiner instead of moving forward with transmission of new content. For large scale, loosely-organized multicast applications, the `NORM_SYNC_CURRENT` policy is typically recommended.

The `NORM_SYNC_ALL` policy allows newly joining receivers much more aggressive behavior as they will immediately NACK for all objects from the "current" object backwards through the entire range of objects set by the `NormSetRxCacheLimit()` function. This behavior depends upon the sender to issue an appropriate `NORM_CMD(SQUELCH)` response (if applicable) to align (i.e. "synchronize") the new receiver with its current transmit object cache (similar to a "repair window"). This synchronization behavior may be useful for unicast uses of NORM or other applications where the group membership is more carefully managed and it is important that all content (including older content) is received. Note that the sender transmit cache bounds (see `NormSetTxCacheBounds()`) and the receiver receive cache limit (see `NormSetRxCacheLimit()`) settings will limit how far back onto the sender transmission history that transmitted objects can be reliably recovered from the "current" transmission point when the receiver begins reception.

When this function is not invoked, the `NORM_SYNC_CURRENT` behavior is observed as the default receiver synchronization policy. This call SHOULD be made before `NormStartReceiver()` is called.

4.5.8.3. Return Values

This function has no return values.

4.5.9. NormSetDefaultNackingMode()

4.5.9.1. Synopsis

```
#include <normApi.h>

void NormSetDefaultNackingMode(NormSessionHandle sessionHandle,
                               NormNackingMode   nackingMode);
```

4.5.9.2. Description

This function sets the default "nacking mode" used when receiving objects for the given *`sessionHandle`*. This allows the receiver application some control of its degree of participation in the repair process. By limiting receivers to only request repair of objects in which they are really interested in receiving, some overall savings in unnecessary network loading might be realized for some applications and users. Available nacking modes include:

`NORM_NACK_NONE`	Do not transmit any repair requests for the newly received object.
`NORM_NACK_INFO_ONLY`	Transmit repair requests for `NORM_INFO` content only as needed.
`NORM_NACK_NORMAL`	Transmit repair requests for entire object as needed.

This function specifies the default behavior with respect to any new sender or object. This default behavior may be overridden for specific sender nodes or specific object using `NormNodeSetNackingMode()` or `NormObjectSetNackingMode()`, respectively. The receiver application's use of `NORM_NACK_NONE` essentially disables a guarantee of reliable reception, although the receiver may still take advantage of sender repair transmissions in response to other receivers' requests. When the sender provides, `NORM_INFO` content for transmitted objects, the `NORM_NACK_INFO_ONLY` mode may allows the receiver to reliably receive object context information from which it may choose to "upgrade" its *`nackingMode`* for the specific object via the `NormObjectSetNackingMode()` call. Similarly, the receiver may changes its default *`nackingMode`* with respect to specific senders via the `NormNodeSetNackingMode()` call. The default "default *`nackingMode`*" when this call is not made is `NORM_NACK_NORMAL`.

4.5.9.3. Return Values

This function has no return values.

4.5.10. NormNodeSetNackingMode()

4.5.10.1. Synopsis

```
#include <normApi.h>

void NormNodeSetNackingMode(NormNodeHandle  nodeHandle,
                            NormNackingMode nackingMode);
```

4.5.10.2. Description

This function sets the default "nacking mode" used for receiving new objects from a specific sender as identified by the *`nodeHandle`* parameter. This overrides the default *`nackingMode`* set for the receive session. See `NormSetDefaultNackingMode()` for a description of possible *`nackingMode`* parameter values and other related information.

4.5.10.3. Return Values

This function has no return values.

4.5.11. NormObjectSetNackingMode()

4.5.11.1. Synopsis

```
#include <normApi.h>

void NormObjectSetNackingMode(NormObjectHandle objectHandle,
                              NormNackingMode  nackingMode);
```

4.5.11.2. Description

This function sets the "nacking mode" used for receiving a specific transport object as identified by the *`objectHandle`* parameter. This overrides the default nacking mode set for the applicable sender node. See `NormSetDefaultNackingMode()` for a description of possible *`nackingMode`* parameter values and other related information.

4.5.11.3. Return Values

This function has no return values.

4.5.12. NormSetDefaultRepairBoundary()

4.5.12.1. Synopsis

```
#include <normApi.h>

void NormSetDefaultRepairBoundary(NormSessionHandle  sessionHandle,
                                  NormRepairBoundary repairBoundary);
```

4.5.12.2. Description

This function allows the receiver application to customize, for a given *`sessionHandle`*, at what points the receiver initiates the NORM NACK repair process during protocol operation. Normally, the NORM receiver initiates NACKing for repairs at the FEC code block and transport object boundaries. For smaller block sizes, the NACK repair process is often/quickly initiated and the repair of an object will occur, as needed, during the transmission of the object. This default operation corresponds to *`repairBoundary`* equal to `NORM_BOUNDARY_BLOCK`. Using this function, the application may alternatively, setting *`repairBoundary`* equal to `NORM_BOUNDARY_OBJECT`, cause the protocol to defer NACK process initiation until the current transport object has been completely transmitted.

4.5.12.3. Return Values

This function has no return values.

4.5.13. NormNodeSetRepairBoundary()

4.5.13.1. Synopsis

```
#include <normApi.h>

void NormNodeSetRepairBoundary(NormNodeHandle     nodeHandle,
                               NormRepairBoundary repairBoundary);
```

4.5.13.2. Description

This function allows the receiver application to customize, for the specific remote sender referenced by the *`nodeHandle`* parameter, at what points the receiver initiates the NORM NACK repair process during protocol operation. See the description of `NormSetDefaultRepairBoundary()` for further details on the impact of setting the NORM receiver repair boundary and possible values for the *`repairBoundary`* parameter.

4.5.13.3. Return Values

This function has no return values.

4.5.14. NormSetDefaultRxRobustFactor()

4.5.14.1. Synopsis

```
#include <normApi.h>

void NormSetDefaultRxRobustFactor(NormSessionHandle sessionHandle,
                                  int               rxRobustFactor);
```

4.5.14.2. Description

This routine controls how persistently NORM receivers will maintain state for sender(s) and continue to request repairs from the sender(s) even when packet reception has ceased. The *`rxRobustFactor`* value determines how many times a NORM receiver will self-initiate NACKing (repair requests) upon cessation of packet reception from a sender. The default value is `20`. Setting *`rxRobustFactor`* to `-1` will make the NORM receiver infinitely persistent (i.e., it will continue to NACK indefinitely as long as it is missing data content). It is important to note that the

`NormSetTxRobustFactor()` also affects receiver operation in setting the time interval that is used to gauge that sender packet transmission has ceased (i.e., the sender inactivity timeout). This "timeout" interval is a equal of (`2 * GRTT * txRobustFactor`). Thus the overall timeout before a NORM receiver quits NACKing is (`rxRobustFactor * 2 * GRTT * txRobustFactor`).

The `NormNodeSetRxRobustFactor()` function can be used to control this behavior on a per-sender basis. When a new remote sender is detected, the default *`rxRobustFactor`* set here is used. Again, the default value is `20`.

4.5.14.3. Return Values

This function has no return values.

4.5.15. NormNodeSetRxRobustFactor()

4.5.15.1. Synopsis

```
#include <normApi.h>

void NormNodeSetRxRobustFactor(NormNodeHandle nodeHandle,
                               int            rxRobustFactor);
```

4.5.15.2. Description

This routine sets the *`rxRobustFactor`* as described in `NormSetDefaultRxRobustFactor()` for an individual remote sender identified by the *`nodeHandle`* parameter. See the description of `NormSetDefaultRxRobustFactor()` for details

4.5.15.3. Return Values

This function has no return values.

4.5.16. NormStreamRead()

4.5.16.1. Synopsis

```
#include <normApi.h>

bool NormStreamRead(NormObjectHandle streamHandle,
                    char*            buffer
                    unsigned int*    numBytes);
```

4.5.16.2. Description

This function can be used by the receiver application to read any available data from an incoming NORM stream. NORM receiver applications "learn" of available NORM streams via `NORM_RX_OBJECT_NEW` notification events. The *`streamHandle`* parameter here must correspond to a valid `NormObjectHandle` value provided during such a prior `NORM_RX_OBJECT_NEW` notification. The *`buffer`* parameter must be a pointer to an array where the received data can be stored of a length as referenced by the *`numBytes`* pointer. On successful completion, the *`numBytes`* storage will be modified to indicate the actual number of bytes copied into the provided *`buffer`*. If the *`numBytes`* storage is modified to a zero value, this indicates that no stream data was currently available for reading.

Note that `NormStreamRead()` is never a blocking call and only returns failure (`false`) when a break in the integrity of the received stream occurs. The `NORM_RX_OBJECT_UPDATE` provides an indication to when there is stream data available for reading. When such notification occurs, the application should repeatedly read from the stream until the *`numBytes`* storage is set to zero, even if a `false` value is returned. Additional `NORM_RX_OBJECT_UPDATE` notifications might not be posted until the application has read all available data.

4.5.16.3. Return Values

This function normally returns a value of `true`. However, if a break in the integrity of the reliable received stream occurs (or the stream has been ended by the sender), a value of `false` is returned to indicate the break. Unless the stream has been ended (and the receiver application will receive `NORM_RX_OBJECT_COMPLETED` notification for the stream in that case), the application may continue to read from the stream as the NORM protocol will automatically "resync" to streams, even if network conditions are sufficiently poor that breaks in reliability occur. If such a "break" and "resync" occurs, the application may be able to leverage other NORM API calls such as `NormStreamSeekMsgStart()` or `NormStreamGetReadOffset()` if needed to recover its alignment with received stream content. This depends upon the nature of the application and its stream content.

4.5.17. NormStreamSeekMsgStart()

4.5.17.1. Synopsis

```
#include <normApi.h>

bool NormStreamSeekMsgStart(NormObjectHandle streamHandle);
```

4.5.17.2. Description

This function advances the read offset of the receive stream referenced by the *`streamHandle`* parameter to align with the next available message boundary. Message boundaries are defined by the sender application using the `NormStreamMarkEom()` call. Note that any received data prior to the next message boundary is discarded by the NORM protocol engine and is not available to the application (i.e., there is currently no "rewind" function for a NORM stream). Also note this call cannot be used to skip messages. Once a valid message boundary is found, the application must read from the stream using `NormStreamRead()` to further advance the read offset. The current offset (in bytes) for the stream can be retrieved via `NormStreamGetReadOffset()`.

4.5.17.3. Return Values

This function returns a value of `true` when start-of-message is found. The next call to `NormStreamRead()` will retrieve data aligned with the message start. If no new message boundary is found in the buffered receive data for the stream, the function returns a value of `false`. In this case, the application should defer repeating a call to this function until a subsequent `NORM_RX_OBJECT_UPDATE` notification is posted.

4.5.18. NormStreamGetReadOffset()

4.5.18.1. Synopsis

```
#include <normApi.h>

unsigned long NormStreamGetReadOffset(NormObjectHandle streamHandle);
```

4.5.18.2. Description

This function retrieves the current read offset value for the receive stream indicated by the *`streamHandle`* parameter. Note that for very long-lived streams, this value may wrap. Thus, in general, applications should not be highly dependent upon the stream offset, but this feature may be valuable for certain applications which associate some application context with stream position.

4.5.18.3. Return Values

This function returns the current read offset in bytes. The return value is undefined for sender streams. There is no error result.

4.6. NORM Object Functions

The functions described in this section may be used for sender or receiver purposes to manage transmission and reception of NORM transport objects. In most cases, the receiver will be the typical user of these functions to retrieve additional information on newly-received objects. All of these functions require a valid `NormObjectHandle` argument which specifies the applicable object. Note that `NormObjectHandle` values obtained from a `NormEvent` notification may be considered valid only until a subsequent call to `NormGetNextEvent()`, unless explicitly retained by the application (see `NormObjectRetain()`). `NormObjectHandle` values obtained as a result of `NormFileEnqueue()`, `NormDataEnqueue()`, or `NormStreamOpen()` calls can be considered valid only until a corresponding

`NORM_TX_OBJECT_PURGED` notification is posted or the object is dequeued using `NormObjectCancel()`, unless, again, otherwise explicitly retained (see `NormObjectRetain()`).

4.6.1. NormObjectGetType()

4.6.1.1. Synopsis

```
#include <normApi.h>

NormObjectType NormObjectGetType(NormObjectHandle objectHandle);
```

4.6.1.2. Description

This function can be used to determine the object type (`(NORM_OBJECT_DAT`, `NORM_OBJECT_FILE`, or `NORM_OBJECT_STREAM`) for the NORM transport object identified by the *`objectHandle`* parameter. The *`objectHandle`* must refer to a current, valid transport object.

4.6.1.3. Return Values

This function returns the NORM object type. Valid NORM object types include `NORM_OBJECT_DATA`, `NORM_OBJECT_FILE`, or `NORM_OBJECT_STREAM`. A type value of `NORM_OBJECT_NONE` will be returned for an *`objectHandle`* value of `NORM_OBJECT_INVALID`.

4.6.2. NormObjectHasInfo()

4.6.2.1. Synopsis

```
#include <normApi.h>

bool NormObjectHasInfo(NormObjectHandle objectHandle);
```

4.6.2.2. Description

This function can be used to determine if the sender has associated any `NORM_INFO` content with the transport object specified by the *`objectHandle`* parameter. This can even be used before the `NORM_INFO` content is delivered to the receiver and a `NORM_RX_OBJECT_INFO` notification is posted.

4.6.2.3. Return Values

A value of `true` is returned if `NORM_INFO` is (or will be) available for the specified transport object. A value of `false` is returned otherwise.

4.6.3. NormObjectGetInfoLength()

4.6.3.1. Synopsis

```
#include <normApi.h>

unsigned short NormObjectGetInfoLength(NormObjectHandle objectHandle);
```

4.6.3.2. Description

This function can be used to determine the length of currently available `NORM_INFO` content (if any) associated with the transport object referenced by the *`objectHandle`* parameter.

4.6.3.3. Return Values

The length of the `NORM_INFO` content, in bytes, of currently available for the specified transport object is returned. A value of `0` is returned if no `NORM_INFO` content is currently available or associated with the object.

4.6.4. NormObjectGetInfo()

4.6.4.1. Synopsis

```
#include <normApi.h>

unsigned short NormObjectGetInfo(NormObjectHandle         objectHandle,
                                 char*                    buffer,
                                 unsigned short           bufferLen);
```

4.6.4.2. Description

This function copies any `NORM_INFO` content associated (by the sender application) with the transport object specified by *`objectHandle`* into the provided memory space referenced by the buffer parameter. The *`bufferLen`* parameter indicates the length of the buffer space in bytes. If the provided *`bufferLen`* is less than the actual `NORM_INFO` length, a partial copy will occur. The actual length of `NORM_INFO` content available for the specified object is returned. However, note that until a `NORM_RX_OBJECT_INFO` notification is posted to the receive application, no `NORM_INFO` content is available and a zero result will be returned, even if `NORM_INFO` content may be subsequently available. The `NormObjectHasInfo()` call can be used to determine if any `NORM_INFO` content will ever be available for a specified transport object (i.e., determine if the sender has associated any `NORM_INFO` with the object in question).

4.6.4.3. Return Values

The actual length of currently available `NORM_INFO` content for the specified transport object is returned. This function can be used to determine the length of `NORM_INFO` content for the object even if a NULL buffer value and zero *`bufferLen`* is provided. A zero value is returned if `NORM_INFO` content has not yet been received (or is non-existent) for the specified object.

4.6.5. NormObjectGetSize()

4.6.5.1. Synopsis

```
#include <normApi.h>

NormSize NormObjectGetSize(NormObjectHandle objectHandle);
```

4.6.5.2. Description

This function can be used to determine the size (in bytes) of the transport object specified by the *`objectHandle`* parameter. NORM can support large object sizes for the `NORM_OBJECT_FILE` type, so typically the NORM library is built with any necessary, related macros defined such that operating system large file support is enabled (e.g., "`#define _FILE_OFFSET_BITS 64`" or equivalent). The `NormSize` type is defined accordingly, so the application should be built with the same large file support configuration.

For objects of type `NORM_OBJECT_STREAM`, the size returned here corresponds to the stream buffer size set by the sender application when opening the referenced stream object.

4.6.5.3. Return Values

A size of the data content of the specified object, in bytes, is returned. Note that it may be possible that some objects have zero data content, but do have `NORM_INFO` content available.

4.6.6. NormObjectGetBytesPending()

4.6.6.1. Synopsis

```
#include <normApi.h>

NormSize NormObjectGetBytesPending(NormObjectHandle objectHandle);
```

4.6.6.2. Description

This function can be used to determine the progress of reception of the NORM transport object identified by the *`objectHandle`* parameter. This function indicates the number of bytes that are pending reception (I.e., when the object is completely received, "bytes pending" will equal ZERO). This function is not necessarily applicable to objects of type `NORM_OBJECT_STREAM` which do not have a finite size. Note it is possible that this function might also be useful to query the "transmit pending" status of sender objects, but it does not account for pending FEC repair transmissions and thus may not produce useful results for this purpose.

4.6.6.3. Return Values

A number of object source data bytes pending reception (or transmission) is returned.

4.6.7. NormObjectCancel()

4.6.7.1. Synopsis

```
#include <normApi.h>

void NormObjectCancel(NormObjectHandle objectHandle);
```

4.6.7.2. Description

This function immediately cancels the transmission of a local sender transport object or the reception of a specified object from a remote sender as specified by the *`objectHandle`* parameter. The *`objectHandle`* must refer to a currently valid NORM transport object. Any resources used by the transport object in question are immediately freed unless the object has been otherwise retained by the application via the `NormObjectRetain()` call. Unless the application has retained the object in such fashion, the object in question should be considered invalid and the application must not again reference the *`objectHandle`* after this call is made.

If the canceled object is a sender object not completely received by participating receivers, the receivers will be informed of the object's cancellation via the NORM protocol `NORM_CMD`(SQUELCH) message in response to any NACKs requesting repair or retransmission of the applicable object. In the case of receive objects, the NORM receiver will not make further requests for repair of the indicated object, but furthermore, will acknowledge the object as completed with respect to any associated positive acknowledgement requests (see `NormSetWatermark()`).

4.6.7.3. Return Values

This function has no return value.

4.6.8. NormObjectRetain()

4.6.8.1. Synopsis

```
#include <normApi.h>

void NormObjectRetain(NormObjectHandle objectHandle);
```

4.6.8.2. Description

This function "retains" the *`objectHandle`* and any state associated with it for further use by the application even when the NORM protocol engine may no longer require access to the associated transport object. Normally, the application is guaranteed that a given `NormObjectHandle` is valid only while it is being actively transported by NORM (i.e., for sender objects, from the time an object is created by the application until it is canceled by the application or purged (see the `NORM_TX_OBJECT_PURGED` notification) by the protocol engine, or, for receiver objects, from the time of the object's `NORM_RX_OBJECT_NEW` notification until its reception is canceled by the application or a `NORM_RX_OBJECT_COMPLETED` or `NORM_RX_OBJECT_ABORTED` notification is posted). Note that an application may refer to a given object after any related notification until the application makes a subsequent call to `NormGetNextEvent()`.

When the application makes a call to `NormObjectRetain()` for a given *`objectHandle`*, the application may use that *`objectHandle`* value in any NORM API calls until the application makes a call to `NormObjectRelease()` for the given object. Note that the application MUST make a corresponding call to `NormObjectRelease()` for each call it has made to `NormObjectRetain()` in order to free any system resources (i.e., memory) used by that object. Also note that retaining a receive object also automatically retains any state associated with the `NormNodeHandle` corresponding to the remote sender of that receive object so that the application may use NORM node API calls for the value returned by `NormObjectGetSender()` as needed.

4.6.8.3. Return Values

This function has no return value.

4.6.9. NormObjectRelease()

4.6.9.1. Synopsis

```
#include <normApi.h>

void NormObjectRelease(NormObjectHandle objectHandle);
```

4.6.9.2. Description

This function complements the `NormObjectRetain()` call by immediately freeing any resources associated with the given *`objectHandle`*, assuming the underlying NORM protocol engine no longer requires access to the corresponding transport object. Note the NORM protocol engine retains/releases state for associated objects for its own needs and thus it is very unsafe for an application to call `NormObjectRelease()` for an *`objectHandle`* for which it has not previously explicitly retained via `NormObjectRetain()`.

4.6.9.3. Return Values

This function has no return value.

4.6.10. NormFileGetName()

4.6.10.1. Synopsis

```
#include <normApi.h>

bool NormFileGetName(NormObjectHandle objectHandle,
                     char*            nameBuffer,
                     unsigned int     bufferLen);
```

4.6.10.2. Description

This function copies the name, as a `NULL`-terminated string, of the file object specified by the *`objectHandle`* parameter into the *`nameBuffer`* of length *`bufferLen`* bytes provided by the application. The *`objectHandle`* parameter must refer to a valid `NormObjectHandle` for an object of type `NORM_OBJECT_FILE`. If the actual name is longer than the provided *`bufferLen`*, a partial copy will occur. Note that the file name consists of the entire path name of the specified file object and the application should give consideration to operating system file path lengths when providing the *`nameBuffer`*.

4.6.10.3. Return Values

This function returns `true` upon success and `false` upon failure. Possible failure conditions include the *`objectHandle`* does not refer to an object of type `NORM_OBJECT_FILE`.

4.6.11. NormFileRename()

4.6.11.1. Synopsis

```
#include <normApi.h>

```

```
bool NormFileRename(NormObjectHandle objectHandle,
                    const char*      fileName);
```

4.6.11.2. Description

This function renames the file used to store content for the `NORM_OBJECT_FILE` transport object specified by the *`objectHandle`* parameter. This allows receiver applications to rename (or move) received files as needed. NORM uses temporary file names for received files until the application explicitly renames the file. For example, sender applications may choose to use the `NORM_INFO` content associated with a file object to provide name and/or typing information to receivers. The *`fileName`* parameter must be a `NULL`-terminated string which should specify the full desired path name to be used. NORM will attempt to create sub-directories as needed to satisfy the request. Note that existing files of the same name may be overwritten.

4.6.11.3. Return Values

This function returns true upon success and false upon failure. Possible failure conditions include the case where the *`objectHandle`* does not refer to an object of type `NORM_OBJECT_FILE` and where NORM was unable to successfully create any needed directories and/or the file itself.

4.6.12. NormDataAccessData()

4.6.12.1. Synopsis

```
#include <normApi.h>

const char* NormDataAccessData(NormObjectHandle objectHandle);
```

4.6.12.2. Description

This function allows the application to access the data storage area associated with a transport object of type `NORM_OBJECT_DATA`. For example, the application may use this function to copy the received data content for its own use. Alternatively, the application may establish "ownership" for the allocated memory space using the `NormDataDetachData()` function if it is desired to avoid the copy.

If the object specified by the *`objectHandle`* parameter has no data content (or is not of type `NORM_OBJECT_DATA`), a NULL value may be returned. The application MUST NOT attempt to modify the memory space used by `NORM_OBJECT_DATA` objects during the time an associated *`objectHandle`* is valid. The length of data storage area can be determined with a call to `NormObjectGetSize()` for the same *`objectHandle`* value.

4.6.12.3. Return Values

This function returns a pointer to the data storage area for the specified transport object. A NULL value may be returned if the object has no associated data content or is not of type `NORM_OBJECT_DATA`.

4.6.13. NormDataDetachData()

4.6.13.1. Synopsis

```
#include <normApi.h>

char* NormDataDetachData(NormObjectHandle objectHandle);
```

4.6.13.2. Description

This function allows the application to disassociate data storage allocated by the NORM protocol engine for a receive object from the `NORM_OBJECT_DATA` transport object specified by the *`objectHandle`* parameter. It is important that this function is called after the NORM protocol engine has indicated it is finished with the data object (i.e., after a `NORM_TX_OBJECT_PURGED`, `NORM_RX_OBJECT_COMPLETED`, or `NORM_RX_OBJECT_ABORTED` notification event). But the application must call `NormDataDetachData()` before a call is made to `NormObjectCancel()` or `NormObjectRelease()` for the object if it plans to access the data content afterwards. Otherwise, the NORM protocol engine will free the applicable memory space when the associated `NORM_OBJECT_DATA` transport object is deleted and the application will be unable to access the received data unless it has previously copied the content.

Once the application has used this call to "detach" the data content, it is the application's responsibility to subsequently free the data storage space as needed.

4.6.13.3. Return Values

This function returns a pointer to the data storage area for the specified transport object. A `NULL` value may be returned if the object has no associated data content or is not of type `NORM_OBJECT_DATA`.

4.6.14. NormObjectGetSender()

4.6.14.1. Synopsis

```
#include <normApi.h>

NormNodeHandle NormObjectGetSender(NormObjectHandle objectHandle);
```

4.6.14.2. Description

This function retrieves the `NormNodeHandle` corresponding to the remote sender of the transport object associated with the given *`objectHandle`* parameter. Note that the returned `NormNodeHandle` value is only valid for the same period that the *`objectHandle`* is valid. The returned `NormNodeHandle` may optionally be retained for further use by the application using the `NormNodeRetain()` function call. The returned value can be used in the NORM Node Functions described later in this document.

4.6.14.3. Return Values

This function returns the `NormNodeHandle` corresponding to the remote sender of the transport object associated with the given *`objectHandle`* parameter. A value of `NORM_NODE_INVALID` is returned if the specified *`objectHandle`* references a locally originated, sender object.

4.7. NORM Node Functions

The functions described in this section may be used for NORM sender or receiver (most typically receiver) purposes to retrieve additional information about a remote *NormNode*, given a valid `NormNodeHandle`. Note that, unless specifically retained (see `NormNodeRetain()`), a `NormNodeHandle` provided in a `NormEvent` notification should be considered valid only until a subsequent call to `NormGetNextEvent()` is made. `NormNodeHandle` values retrieved using `NormObjectGetSender()` can be considered valid for the same period of time as the corresponding `NormObjectHandle` is valid.

4.7.1. NormNodeGetId()

4.7.1.1. Synopsis

```
#include <normApi.h>

NormNodeId NormNodeGetId(NormNodeHandle nodeHandle);
```

4.7.1.2. Description

This function retrieves the `NormNodeId` identifier for the remote participant referenced by the given *`nodeHandle`* value. The `NormNodeId` is a 32-bit value used within the NORM protocol to uniquely identify participants within a NORM session. The participants identifiers are assigned by the application or derived (by the NORM API code) from the host computers default IP address.

4.7.1.3. Return Values

This function returns the `NormNodeId` value associated with the specified *`nodeHandle`*. In the case *`nodeHandle`* is equal to `NORM_NODE_INVALID`, the return value will be `NORM_NODE_NONE`.

4.7.2. NormNodeGetAddress()

4.7.2.1. Synopsis

```
#include <normApi.h>

bool NormNodeGetAddress(NormNodeHandle          nodeHandle,
                        char*                   addrBuffer,
                        unsigned int*           bufferLen,
                        unsigned short*         port = NULL);
```

4.7.2.2. Description

This function retrieves the current network source address detected for packets received from remote NORM sender referenced by the *`nodeHandle`* parameter. The *`addrBuffer`* must be a pointer to storage of *`bufferLen`* bytes in length in which the referenced sender node's address will be returned. Optionally, the remote sender source port number (see `NormSetTxPort()`) is also returned if the optional port pointer to storage parameter is provided in the call. Note that in the case of Network Address Translation (NAT) or other firewall activities, the source address detected by the NORM receiver may not be the original address of the original NORM sender.

4.7.2.3. Return Values

A value of `true` is returned upon success and `false` upon failure. An invalid *`nodeHandle`* parameter value would lead to such failure.

4.7.3. NormNodeGetGrtt()

4.7.3.1. Synopsis

```
#include <normApi.h>

double NormNodeGetId(NormNodeHandle nodeHandle);
```

4.7.3.2. Description

This function retrieves the advertised estimate of group round-trip timing (GRTT) for the remote sender referenced by the given *`nodeHandle`* value. Newly-starting senders that have been participating as a receiver within a group may wish to use this function to provide a more accurate startup estimate of GRTT (see `NormSetGrttEstimate()`) prior to a call to `NormStartSender()`. Applications may use this information for other purpose as well. Note that the `NORM_GRTT_UPDATED` event is posted (see `NormGetNextEvent()`) by the NORM protocol engine to indicate when changes in the local sender or remote senders' GRTT estimate occurs.

4.7.3.3. Return Values

This function returns the remote sender's advertised GRTT estimate in units of seconds. A value of `-1.0` is returned upon failure. An invalid *`nodeHandle`* parameter value will lead to such failure.

4.7.4. NormNodeGetCommand()

4.7.4.1. Synopsis

```
#include <normApi.h>

bool NormNodeGetCommand(NormNodeHandle nodeHandle,
                        char*          buffer,
                        unsigned int*  buflen);
```

4.7.4.2. Description

This function retrieves the content of an application-defined command that was received from a remote sender associated with the given *`nodeHandle`*. This call should be made in response to the `NORM_RX_CMD_NEW` notification. This notification is issued for each command received. However the application may use this call to poll for received commands if desired. Additionally, the received command length can be "queried" by setting the value referenced by the *`buflen`* parameter to `ZERO`. Upon return, this value referenced by the *`buflen`* parameter is adjusted to reflect the command length. Then a subsequent call to `NormNodeGetCommand()` can be made with an appropriately-sized buffer to retrieve the received command content. The command size will be less than or equal to the NORM segment size configured for the given remote sender.

Note that if a rapid succession of commands are sent it is possible that the commands may be delivered to the receivers out-of-order. Also, when repetition is requested (i.e., if *`robust`* is set to `true`) the receiver may receive duplicate copies of the same command. It is up to the application to provide any needed mechanism for detecting and/or filtering duplicate command reception.

4.7.4.3. Return Values

This function returns `true` upon successful retrieval of command content. A return value of `false` indicates that either no command was available or the provided buffer size (*`buflen`* parameter) was inadequate. The value referenced by the *`buflen`* parameter is adjusted to indicate the actual command length (in bytes) upon return.

4.7.5. NormNodeFreeBuffers()

4.7.5.1. Synopsis

```
#include <normApi.h>

void NormNodeFreeBuffers(NormNodeHandle nodeHandle);
```

4.7.5.2. Description

This function releases memory resources that were allocated for a remote sender. For example, the receiver application may wish to free memory resources when receiving a `NORM_REMOTE_SENDER_INACTIVE` notification for a given remote sender when multiple senders may be providing content. The NORM protocol engine allocates memory for reliable transport buffering on a per sender basis according to the limit set in the `NormStartReceiver()` call. These buffering resources comprise the majority of the state allocated for a given remote sender. For NORM applications with possibly multiple senders active at different times, this function can be used to manage to amount of memory allocated for reliable reception. If a sender becomes "active" again after a call to `NormNodeFreeBuffers()`, new memory resources will be allocated. Note that state for any pending (uncompleted) objects will be dropped when this function is called and the receiver may request retransmission and repair of content if the sender once again becomes "active". The application SHOULD call `NormObjectCancel()` for any pending objects *before* calling `NormNodeFreeBuffers()` if it wishes to never receive those pending objects. Alternatively, a call to `NormNodeDelete()` will completely eliminate all state for a given remote sender and, if that sender becomes "active" again, it will be treated as a completely new sender.

4.7.5.3. Return Values

This function has no return value.

4.7.6. NormNodeDelete()

4.7.6.1. Synopsis

```
#include <normApi.h>

void NormNodeDelete(NormNodeHandle nodeHandle);
```

4.7.6.2. Description

This function can be used by a NORM receiver application to completely remove the state associated with a remote sender for the given *`nodeHandle`*. For example, when a `NORM_REMOTE_SENDER_INACTIVE` notification occurs for a given sender, the application may wish to completely free *all* associated resources. Note this is distinct from the `NormNodeFreeBuffers()` call where only the buffering resources are freed and other state pertaining to the sender is kept. If the deleted sender again becomes "active", it will be treated as a brand new sender. Unless explicitly retained with a call to `NormNodeRetain()`, the *`nodeHandle`* should be considered invalid after this call is made.

Additionally, any NormObjectHandle values for pending objects from this sender are also invalidated (unless otherwise retained), although `NORM_RX_OBJECT_ABORTED` notifications may be issued for those pending objects.

4.7.6.3. Return Values

This function has no return value.

4.7.7. NormNodeRetain()

4.7.7.1. Synopsis

```
#include <normApi.h>

void NormNodeRetain(NormNodeHandle nodeHandle);
```

4.7.7.2. Description

In the same manner as the `NormObjectRetain()` function, this function allows the application to retain state associated with a given *`nodeHandle`* value even when the underlying NORM protocol engine might normally free the associated state and thus invalidate the `NormNodeHandle`. If the application uses this function, it must make a corresponding call to `NormNodeRelease()` when finished with the node information to avoid a memory leak condition. `NormNodeHandle` values (unless retained) are valid from the time of a `NORM_REMOTE_SENDER_NEW` notification until a complimentary `NORM_REMOTE_SENDER_PURGED` notification. During that interval, the application will receive `NORM_REMOTE_SENDER_ACTIVE` and `NORM_REMOTE_SENDER_INACTIVE` notifications according to the sender's message transmission activity within the session.

It is important to note that, if the NORM protocol engine posts a `NORM_REMOTE_SENDER_PURGED` notification for a given `NormNodeHandle`, the NORM protocol engine could possibly, subsequently establish a new, different `NormNodeHandle` value for the same remote sender (i.e., one of equivalent `NormNodeId`) if it again becomes active in the session. A new `NormNodeHandle` may likely be established even if the application has retained the previous `NormNodeHandle` value. Therefore, to the application, it might appear that two different senders with the same `NormNodeId` are participating if these notifications are not carefully monitored. This behavior is contingent upon how the application has configured the NORM protocol engine to manage resources when there is potential for a large number of remote senders within a session (related APIs are TBD). For example, the application may wish to control which specific remote senders for which it keeps state (or limit the memory resources used for remote sender state, etc) and the NORM API may be extended in the future to control this behavior.

4.7.7.3. Return Values

This function has no return value.

4.7.8. NormNodeRelease()

4.7.8.1. Synopsis

```
#include <normApi.h>

void NormNodeRelease(NormNodeHandle nodeHandle);
```

4.7.8.2. Description

In complement to the `NormNodeRetain()` function, this API call releases the specified *`nodeHandle`* so that the NORM protocol engine may free associated resources as needed. Once this call is made, the application should no longer reference the specified `NormNodeHandle`, unless it is still valid.

4.7.8.3. Return Values

This function has no return value.

4.8. NORM Debugging Functions

This section describes some additional function calls that are available to set debugging output options and control other aspects of the NORM implementation.

4.8.1. NormSetDebugLevel()

4.8.1.1. Synopsis

```
#include <normApi.h>

void NormSetDebugLevel(unsigned int level);
```

4.8.1.2. Description

This function controls the verbosity of NORM debugging output. Higher values of level result in more detailed output. The highest level of debugging is 12. The debug output consists of text written to STDOUT by default but may be directed to a log file using the `NormOpenDebugLog()` function.

4.8.1.3. Return Values

This function has no return value.

4.8.2. NormOpenDebugLog()

4.8.2.1. Synopsis

```
#include <normApi.h>

bool NormOpenDebugLog(NormInstanceHandle instance, const char* fileName);
```

4.8.2.2. Description

This function allows NORM debug output to be directed to a file instead of the default `STDERR`.

4.8.2.3. Return Values

The function returns `true` on success. If the specified file cannot be opened a value of `false` is returned.

4.8.3. NormCloseDebugLog()

4.8.3.1. Synopsis

```
#include <normApi.h>

bool NormCloseDebugLog(NormInstanceHandle instance);
```

4.8.3.2. Description

TBD

4.8.3.3. Return Values

TBD

4.8.4. NormOpenDebugPipe()

4.8.4.1. Synopsis

```
#include <normApi.h>

bool NormOpenDebugPipe(NormInstanceHandle instance, const char* pipeName);
```

4.8.4.2. Description

TBD

4.8.4.3. Return Values

TBD

4.8.5. NormCloseDebugPipe()

4.8.5.1. Synopsis

```
#include <normApi.h>

bool NormCloseDebugPipe(NormInstanceHandle instance);
```

4.8.5.2. Description

TBD

4.8.5.3. Return Values

TBD