Channels

Table of Contents

- Description
- Example

Description

Dragon Channels is the low-level transport layer for communicating messages between POSIX processes in the Dragon runtime. Channels are a thread and interprocess-safe queue where messages can be sent and received. The Dragon run-time services itself uses channels for communication between processes. User programs, utilizing the Dragon run-time services also use channels either directly, or indirectly. This API would be used directly when a program wants to communicate in the most efficient way possible while being willing to give up the services provided by higher level communication and synchronization abstractions. Choosing to communicate at the channels level will mean giving up features like pickling and unpickling, automatic serialization of data, and the automatic streaming of data over a channel.

Channels provide flexible on-node and communication that processes use by attaching and detaching to the underlying Managed Memory. When Channels are used in conjunction with a transport agent, transparent off-node communication is also provided when sending and receiving messages using the Channels API. In this respect, Channels resemble POSIX sockets as they are always available as a service and not built upon the static distributed model that MPI or SHMEM is. Transparency is provided because the exact same Channels API calls work for both on-node and off-node communication. The user program does not change when communicating off-node or on-node.

A major advantage of Channels is that they retain the flexibility of using sockets or a filesystem while enabling zero-copy on-node transfers, single-copy RDMA-enabled transfers off-node, and choices for blocking semantics. There is a rich set of buffer management options that enable use cases such as:

- static target-side managed payload landing pads much like SHMEM or one-sided MPI
- dynamic target-side managed payload landing pads much like two-sided MPI
- static or dynamic origin-side managed payload landing pad, which nothing else has

Dragon Channels can reside in any valid `dragonMemoryDescr_t` as provided by Dragon Memory Pools. This includes shared memory, a filesystem, or private virtual memory. Payload buffers for messages can reside within a channel, a memory pool `dragonMemoryPoolDescr_t` the channel was allocated from, or any valid `dragonMemoryDescr_t` passed with a message. This design allows Channels to provide multiple usage scenarios with different performance and persistence characteristics.

Channels provide both blocking an non-blocking semantics for sending and getting messages. When sending, blocking can be chosen to wait when memory from a pool is needed and not available. When getting messages, blocking receives will wait until a message is available. Blocking operations can either idle wait, consuming fewer resources and energy, or spin wait, with relatively no wakeup cycles. Channels are highly configurable and customizable for whatever situation they are used in at the expense of being a low-level primitive synchronization and communication construct.

Enums

enum **dragonChannelOFlag_t**

Open mode with existing channel.

Not yet implemented.

Values:

enumerator **DRAGON_CHANNEL_EXCLUSIVE**

enumerator **DRAGON_CHANNEL_NONEXCLUSIVE**

enum **dragonChannelFC_t**

Flow control options.

Not yet implemented

Values:

enumerator **DRAGON_CHANNEL_FC_NONE**

enumerator **DRAGON_CHANNEL_FC_RESOURCES**

enumerator **DRAGON_CHANNEL_FC_MEMORY**

enumerator **DRAGON_CHANNEL_FC_MSGS**

enum dragonChannelEvent_t

Channel Poll Event Mask.

This defines the valid poll operations. Combinations of the values below are only allowed when explicitly given in these constants. The POLLINOUT combination is allowed, but all other constants are valid only individually for channel poll operations.

Values:

enumerator **DRAGON_CHANNEL_POLLNOTHING**

Used for resetting an event mask. Setting an event mask to this value clears all flags.

enumerator **DRAGON_CHANNEL_POLLIN**

Wait for available message. If a timeout occurs, then poll will return with an appropriate return code. Otherwise, upon successful completion, a message arrived in the channel. NOTE: Due to the nature of channels, another process/thread may have already retrieved the message by the time the current process attempts to receive it.

enumerator **DRAGON_CHANNEL_POLLOUT**

Wait for available space in channel. If no timeout occurs, the poll operation returns when there is space in the channel. As with POLLIN, while space is available when poll returns from POLLOUT the space may be gone again before this process can send to the channel if multiple processes are sending to the channel simultaneously.

enumerator **DRAGON_CHANNEL_POLLINOUT**

Get notified of available messages or available space in channel. This effectively means that a process is notified of any changes to a channel.

enumerator **DRAGON_CHANNEL_POLLEMPTY**

Check that channel is empty. The poll operation returns when the channel becomes empty or when a timeout occurrs. If the channel is empty when poll is called it returns immediately.

enumerator **DRAGON_CHANNEL_POLLFULL**

Check whether channel is full. The poll operation returns when the channel becomes full or when a timeout occurs. If the channel is full when poll is called it returns immediately.

enumerator **DRAGON_CHANNEL_POLLSIZE**

Get the number of messages in the channel. In this case, unlike other calls to poll, the number of messages currently in the channel are return in place of the return code.

enumerator **DRAGON_CHANNEL_POLLRESET**

Resets the channel, immediately deleting all messages. The deleted messages are immediately cleared with no cleanup. The message contents are not freed, so resetting a channel does not change the reference counts of any messages or memory that might have been in the channel at the time of this call.

enumerator **DRAGON_CHANNEL_POLLBARRIER_WAIT**

When channel is used as a barrier, wait on barrier with this. Using the channel barrier support is further discussed in the channels description.

enumerator **DRAGON_CHANNEL_POLLBARRIER_ABORT**

When channel is used as a barrier, abort all waiters on barrier.

enumerator **DRAGON_CHANNEL_POLLBARRIER_RELEASE**

When channel is used as a barrier, release all barrier waiters.

enum **dragonChannelFlags_t**

Channel flags for Testing.

These flags are for test purposes only.

Values:

enumerator **DRAGON_CHANNEL_FLAGS_NONE**

enumerator **DRAGON_CHANNEL_FLAGS_MASQUERADE_AS_REMOTE**

this is a process local attribute only

enum **dragonChannelRecvNotif_t**

Receive Notification Type.

Not yet implemented.

Values:

enumerator **DRAGON_RECV_SYNC_SIGNAL**

enumerator **DRAGON_RECV_SYNC_MANUAL**

enum **dragonGatewayMessageKind_t**

Types of GatewayMessages.

This structure defines the types of GatewayMessages that can be ceated and must be supported by a transport agent.

Values:

enumerator **DRAGON_GATEWAY_MESSAGE_SEND**

A send operation

enumerator **DRAGON_GATEWAY_MESSAGE_GET**

A get operation

enumerator **DRAGON_GATEWAY_MESSAGE_EVENT**

An event monitoring operation

Functions

dragonError_t **dragon_channel_create**(dragonChannelDescr_t *ch, *const* dragonC_UID_t c_uid, dragonMemoryPoolDescr_t *pool_descr, *const* dragonChannelAttr_t *attr)

Create a channel in a memory pool.

Create a channel in a memory pool with the given c_uid. While the c_uid is not enforced to be unique across the Dragon run-time services by this api, that is desirable and is left to the user of this API call. Unique c_uid values are enforced by this API call at the process level. If you need globally unique c_uids, then use the managed channel API instead which coordinates with the Dragon run-time services to insure global uniqueness. For transparent multi-node communication between channels, c_uids do not have to be unique across all nodes.

Parameters:

- **ch** – is the channel descriptor which will be initialzed from this call.
- **c_uid** – is a channel identifier. It must be unique on a process level and should be unique across all nodes. Read about uniqueness in the general description.
- **pool_descr** – is the pool in which to allocate this channel.
- **attr** – are the attributes to be used in creating this channel. If providing attributes, make sure you call dragon_channel_attr_init first. Otherwise, NULL can be provided to get default attributes.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_attr_init**(dragonChannelAttr_t *attr)

Initialize a channel attributes structure.

When custom channel attributes are desired, this function should be called first to initialize them to default values. Then the user may override desired attributes before using it in creating a channel.

Parameters: **attr** – is a pointer to the channel attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_attr_destroy**(dragonChannelAttr_t *attr)

Destroy a channel attributes structure.

Release any allocations in the attributes structure. This does not destroy the underlying channel.

Parameters: **attr** – is a pointer to the attributes structure that was previously inited.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_get_attr**(*const* dragonChannelDescr_t *ch, dragonChannelAttr_t *attr)

Get the attributes of a channel.

Calling this will initialize a channel attributes structure with read-only and user-specified attributes of the channel. Read-only attributes are supplied on this call so a caller may inspect the current state of the channel.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **attr** – is a pointer to a channel attributes structure that will be initialized by this call.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_destroy**(dragonChannelDescr_t *ch)

Destroy a channel.

Destroying a channel can only be done on the node where the channel is located. Destroying a channel frees the allocated memory in the memory pool and invalidates any use of the channel from this or other processes.

Parameters: **ch** – is the channel descriptor to destroy.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_serialize**(*const* dragonChannelDescr_t *ch, dragonChannelSerial_t *ch_ser)

Serialize a channel to be shared with another process.

Parameters:
- **ch** – is a channel descriptor for a channel.
- **ch_ser** – is a serialized channel descriptor that may be shared with another process.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_serial_free**(dragonChannelSerial_t *ch_ser)

Free the internal resources of a serialized channel descriptor.

This frees internal structures of a serialized memory descriptor. It does not destroy the channel itself.

Parameters: **ch_ser** – is a serialized channel descriptor.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_attach**(*const* dragonChannelSerial_t *ch_ser, dragonChannelDescr_t *ch)

Attach to a channel.

Calling this attaches to a channel by using a serialized channel descriptor that was passed to this process. The serialized channel descriptor must have been created using the dragon_channel_serialize function.

Parameters:
- **ch_ser** – is a pointer to the serialized channel descriptor.
- **ch** – is a pointer to a channel descriptor that will be initialized by this call.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_detach**(dragonChannelDescr_t *ch)

Detach from a channel.

Calling this will clean up any local references to a channel and release any local resources needed for communicating with the channel. It does not destroy the channel itself.

Parameters: **ch** – is the channel descriptor from which to detach.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_get_pool**(*const* dragonChannelDescr_t *ch, dragonMemoryPoolDescr_t *pool_descr)

Get pool from channel.

Given a channel, get the pool from the channel where it resides.

Parameters:
- **ch** – is a pointer to the channel descriptor.
- **pool_descr** – is a pointer the the pool descriptor that will be initialized by this call.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_get_hostid**(*const* dragonChannelDescr_t *ch, dragonULInt *hostid)

Get the host identifier for a channel.

The host identifier is used to decide on which node the channel is hosted. This determines if it is a local channel or a remote channel. Sends and receives to/from remote channels are handled through gateway channels that must be registered prior to sending and or receiving. Gateway channels are shared with a process via an environment variable or variables. See register_gateways_from_env for more information on registering gateway channels. Host ids are arbitrary unsigned long integers and cannot not be interpreted or inferred from any other data.

Parameters:
- **ch** – is an initialized channel descriptor
- **hostid** – is a pointer to space to provide the hostid.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_get_uid_type**(*const* dragonChannelSerial_t *ch_ser, dragonULInt *cuid, dragonULInt *type)

Get the channel's cuid and/or type.

From a serialized channel descriptor, this function will return the cuid of the channel and the type of the channel. The channel is not attached while doing this.

Parameters:
- **ch_ser** – is a pointer to a serialied channel descriptor.
- **cuid** – is a pointer to a location there the cuid will be stored. If NULL is provided, the cuid is not copied from the channel descriptor.
- **type** – is a pointer to a location where the channel's type will be stored. If NULL is provided, the type is not copied from the channel descriptor.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_pool_get_uid_fname**(*const* dragonChannelSerial_t *ch_ser, dragonULInt *muid, char **pool_fname)

Get the muid and filename of a channel's pool.

From a serialized channel descriptor, this function returns information about the pool the channel is allocated in, including the muid of the pool and the pool's filename. The channel is not attached while doing this.

Parameters:
- **ch_ser** – is a pointer to a serialized channel descriptor.
- **muid** – is a pointer to a location where the muid of the pool will be stored. If NULL is provided, then the muid is not copied from the channel.
- **pool_fname** – is the filename associated with the shared memory of the channel's pool.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_send_attr_init**(dragonChannelSendAttr_t *send_attr)

Initialize a send attributes structure.

When custom send attributes are desired, this function should be called first, to initialize the to default values. Then the user may override desired attributes before using it in creating a send handle.

Parameters: **send_attr** – is a pointer to the send attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_recv_attr_init**(dragonChannelRecvAttr_t *recv_attr)

Initialize a receive attributes structure.

When custom receive attributes are desired, this function should be called first to initialize them to default values. Then the user may override desired attributes before using it in creating a receive handle.

Parameters: **recv_attr** – is a pointer to the receive attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_send_attr_destroy**(dragonChannelSendAttr_t *send_attr)

Destroy a channel send handle attributes structure.

Release any allocations or resources associated with a send handle attributes structure. This does not destroy the underlying channel.

Parameters: **send_attr** – is a pointer to the channel send handle attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_recv_attr_destroy**(dragonChannelRecvAttr_t *recv_attr)

Destroy a channel recv handle attributes structure.

Release any allocations or resources associated with a recv handle attributes structure. This does not destroy the underlying channel.

Parameters: **recv_attr** – is a pointer to the channel send handle attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_sendh**(*const* dragonChannelDescr_t *ch, dragonChannelSendh_t *ch_sh, *const* dragonChannelSendAttr_t *attr)

Initialize a send handle on a channel.

Calling this initializes a send handle on a channel. To use the send handle it must also be opened using the dragon_chsend_open function. Messages sent with a single send handle are guaranteed to be received from the channel in the same order. Order is maintained by both send and receive handles. If customized send handle attributes are to be supplied, dragon_channel_send_attr_init should be called first to initialize all attributes to default values before customizing the desired values.

Parameters:
- **ch** – is an initialized channel descriptor.
- **ch_sh** – is a pointer to a send handle descriptor that will be initialized by this call.
- **attr** – are send handle attributes that may be supplied when initializing the send handle. Providing NULL will result in the default send handle attributes being applied.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chsend_get_attr**(*const* dragonChannelSendh_t *ch_sh, dragonChannelSendAttr_t *attr)

Get the attributes of a send handle.

Copy the attributes from a send handle into a send attributes structure. The handle does not need to be open in order to get the attributes.

Parameters:
- **ch_sh** – A pointer to the channel send handle.
- **attr** – A pointer to the channel send handle attributes structure to copy into.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_recvh**(*const* dragonChannelDescr_t *ch, dragonChannelRecvh_t *ch_rh, *const* dragonChannelRecvAttr_t *attr)

Initialized a channel receive handle.

Initialize a receive handle structure given a channel. To use the receive handle it must also be opened using the dragon_chrecv_open function. Messages received with a single receive handle are guaranteed to be received from the channel in the same order they were sent. Order is maintained by both send and receive handles. If customized receive handle attributes are to be supplied, dragon_channel_recv_attr_init should be called first to initialize all attributes to default values before customizing the desired values.

Parameters:
- **ch** – is an initialized channel descriptor
- **ch_rh** – is a pointer to a structure that will be initalized by this call.
- **rattrs** – is a pointer to receive handle attributes. If NULL is provided, default receive handle attributes will be used.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chrecv_get_attr**(*const* dragonChannelRecvh_t *ch_rh, dragonChannelRecvAttr_t *attr)

Get the attributes for a receive handle.

Copy the attributes from a receive handle into a receive attributes structure. The handle does not need to be open in order to get the attributes.

Parameters:
- **ch_rh** – A pointer to the channel receive handle.
- **attr** – A pointer to the channel receive handle attributes structure to copy into.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_message_attr_init**(dragonMessageAttr_t *attr)

Initialize a message attributes structure.

When creating user-defined send attributes, this function should be called first, to initialize it. Then the user may override desired attributes before using it in creating a send handle.

Parameters: **send_attr** – A pointer to the send attributes structure.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_message_attr_destroy**(dragonMessageAttr_t *attr)

---

dragonError_t **dragon_channel_message_getattr**(*const* dragonMessage_t *msg, dragonMessageAttr_t *attr)

Get Message attributes.

Get the attributes for an existing Message.

Parameters:
- **msg** – is a pointer to the dragonMessage structure.
- **attr** – is a pointer to update with an attributes structure.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_message_setattr**(dragonMessage_t *msg, *const* dragonMessageAttr_t *attr)

Set Message attributes.

Set the attributes for an existing Message.

Parameters:
- **msg** – is a pointer to the dragonMessage structure to update with new attributes.
- **attr** – is a pointer to an attributes structure.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_message_init**(dragonMessage_t *msg, dragonMemoryDescr_t *mem_descr, *const* dragonMessageAttr_t *attr)

Initialize a new message for transfer through Channels.

Create a new Message that either wraps up an existing Managed Memory descriptor or is left empty so it can be later updated with a Managed Memory descriptor.

Parameters:
- **msg** – is a pointer to the dragonMessage structure to update.
- **mem_descr** – is a pointer to a Managed Memory descriptor to wrap up or NULL.
- **mattr** – is a pointer to Message attributes for the new Message or NULL to use default values.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_message_destroy**(dragonMessage_t *msg, *const* bool free_mem_descr)

---

dragonError_t **dragon_channel_message_get_mem**(*const* dragonMessage_t *msg, dragonMemoryDescr_t *mem_descr)

Get a Managed Memory descriptor associated with a Message.

Get a descriptor for the Managed Memory allocation associated with the Message. An error will be returned if the Message has no Managed Memory descriptor associated with it.

Parameters:
- **msg** – is a pointer to the dragonMessage structure.
- **mem_descr** – is a pointer to update with a Managed Memory descriptor.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_message_count**(*const* dragonChannelDescr_t *ch, uint64_t *count)

Get the number of messages in a channel.

This will return the number of messages found in a channel. The number of messages will be correct when queried, but may change before this function returns to the caller if other processes are interacting with the channel by sending or receiving messages.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **count** – is a pointer to space for a 64 bit unsigned integer that will hold the number of messages found in the channel.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

bool **dragon_channel_is_local**(*const* dragonChannelDescr_t *ch)

Check to see is a channel is local or not.

Returns true if a channel is local and false if not.

Parameters: **ch** – is an intialized channel descriptor.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chsend_open**(dragonChannelSendh_t *ch_sh)

Open a channel send handle.

Open a send handle on a channel. Once opened, the send handle can be used for sending messages.

Parameters: **ch_sh** – is a pointer to an initialized send handle.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chsend_close**(dragonChannelSendh_t *ch_sh)

Close a send handle.

Once sending is done on a particular send handle, it should be closed.

Parameters: **ch_sh** – is a pointer to an open send handle.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chsend_send_msg**(*const* dragonChannelSendh_t *ch_sh, *const* dragonMessage_t *msg_send, dragonMemoryDescr_t *dest_mem_descr, *const* timespec_t *timeout)

Send a message into a channel.

Much like putting data in a queue, messages can be sent through channels. Sending a message into a channel can be done in several ways. The destination memory descriptor (i.e. dest_mem_descr) may be NULL in which case the memory associated with the message to send will be copied upon sending. In that way the caller retains ownwership of the messages original memory allocation. If dest_mem_descr is non-NULL, then the message is copied into the dest_mem_descr before sending it. However, if dest_mem_descr is the special value DRAGON_CHANNEL_SEND_TRANSFER_OWNERSHIP, then this indicates the caller wishes to transfer the ownership of the memory allocation in msg_send to the receiver of this message. In this way, a zero-copy send operation is possible when sending on-node between processes. Using transfer of ownership also has performance advantages when communicating between nodes since the path through the transport agent then has fewer copies as well.

If a NULL dest_mem_descr is provided, the memory allocation for the message copy is made from either the channel's pool (if the channel is local) or from the default memory pool for the node where the message is being sent from.

When a memory allocation is required because the dest_mem_descr is NULL, calling this function may block the sender until the memory allocation is available. This call will also block while waiting for exclusive access to the channel. If a timeout occurs on sending, then it was because the required memory was not available.

Parameters:
- **ch_sh** – is a pointer to an initialized and open send handle.
- **msg_send** – is a pointer to a valid channel message descriptor.
- **dest_mem_descr** – is a pointer to a destination memory descriptor. The detailed description above has more details on the valid values for this argument.
- **timeout_override** – is a pointer to a timeout structure that may be used to override the default send timeout as provided in the send handle attributes. A timeout of zero seconds and zero nanoseconds will result in a try-once call of send. A value of NULL will result in using the default timeout from the send handle attributes.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chrecv_open**(dragonChannelRecvh_t *ch_rh)

Open a channel receive handle.

Once opened, a receive handle can be used to receive messages from a channel. Messages received from a receive handle are guaranteed to be received in the order they were sent.

Parameters: **ch_rh** – is the receive handle to be opened.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chrecv_close**(dragonChannelRecvh_t *ch_rh)

Close a channel receive handle.

Once a receive handle is no longer to be used for receiving, the handle should be closed.

Parameters: **ch_rh** – is a pointer to an open channel receive handle.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chrecv_get_msg**(*const* dragonChannelRecvh_t *ch_rh, dragonMessage_t *msg_recv)

Receive a message from a channel.

Calling this receives the next available message from a channel without blocking.

Parameters:
- **ch_rh** – is a pointer to an intialized and open receive handle structure.
- **msg_recv** – is a pointer to a message structure that will be initialized with the received message.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred. If no message is immediately available it will return DRAGON_CHANNEL_EMPTY.

---

dragonError_t **dragon_chrecv_get_msg_notify**(dragonChannelRecvh_t *ch_rh, dragonMessage_t *msg_recv, dragonBCastDescr_t *bd)

Asynchronously receive a message from a channel.

This function is not currently implemented.

Parameters:
- **ch_rh** – is a pointer to an initialized, open receive handle.
- **msg_recv** – is a pointer to a message descriptor to be initialzed aynchronously.
- **sync** – is a pointer to a BCast descriptor which points to a valid BCast object. The calling process can then wait on this BCast for a message to receive. This can be safely done by a thread of the calling process or by the process itself.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chrecv_get_msg_blocking**(*const* dragonChannelRecvh_t *ch_rh, dragonMessage_t *msg_recv, *const* timespec_t *timeout)

Receive a message from a channel.

This receives a message from a channel into the supplied message structure. A timeout override can be supplied to override the default timeout found in the channel receive handle. If the timeout_override is NULL, the default receive handle timeout is used. A timeout of zero seconds and zero nanoseconds will result in try once attempt which will return DRAGON_CHANNEL_EMPTY if no message is available.

Parameters:
- **ch_rh** – is a pointer to an initialized, open receive handle.
- **msg_recv** – is a pointer to a message descriptor which will be initialized by this call upon successful completion should a message be received.
- **timeout_override** – is a pointer to a structure that contains the timeout to use in place of the default handle timeout. If NULL, the default handle timeout is used. If zero seconds and nanoseconds, the call is non-blocking and returns the next message if availble or DRAGON_CHANNEL_EMPTY if no message is currently available.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_poll**(*const* dragonChannelDescr_t *ch, dragonWaitMode_t wait_mode, *const* short event_mask, *const* timespec_t *timeout, dragonULInt *result)

Poll a channel for status or the occurrence of an event.

Polling may be done on any of the poll values found on dragonChannelEvent_t. The polling values POLLIN and POLLOUT may be done together with the value DRAGON_CHANNEL_POLLINOUT. All other polling event_masks must be done separately. They are not supported in combination with each other.

Parameters:
- **ch** – A channel descriptor for a channel either on-node or off-node.
- **wait_mode** – A choice between IDLE waiting or SPIN waiting for events polling. It is only relevant when polling for events.
- **event_mask** – This specifies one of the dragonChannelEvent_t constants.

- **timeout** – NULL indicates blocking with no timeout. Otherwise, blocks for the specified amount of time. If (0,0) is provided, then it is a non-blocking call.
- **result** – For all but the DRAGON_CHANNEL_POLLSIZE the result will be a 64-bit cast of the event_mask field. With DRAGON_CHANNEL_POLLSIZE it is the number of messages currently in the channel.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_add_event_bcast**(dragonChannelDescr_t *ch, dragonBCastSerial_t *ser_bcast, *const* short event_mask, int user_token, dragonULInt *channel_token)

Create a channel set from a list of channels with possible attributes.

You create a channel set when you wish to poll across a set of channels. The details of how this is accomplished is handled by the channel set API. In this call you provide a list of the channels you wish to poll across. To get the most efficient implementation of this multi-channel poll, if it is in your control, specify the same default bcast event descriptor in the channel attributes for each channel when it is created.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **ser_bcast** – is a serialized BCast decriptor to be added to the channel.
- **event_mask** – is the event mask to be monitored in the channel and signaled via the given BCast object.
- **user_token** – is a user-supplied token to be provided when the BCast is triggered.
- **channel_token** – is a pointer to space for an unsigned integer that is returned as a channel identifier when the event is triggered for this channel. This is used by the channel itself, not by the caller, but is provided for subsequent event bcast calls on this channel.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_remove_event_bcast**(dragonChannelDescr_t *ch, dragonULInt channel_token)

Remove an event BCast from this channel.

The channel_token returned when a process adds an event bcast to a channel can be used to subsequently delete the even bcast from the channel.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **channel_token** – is the channel token that was provided when the event bcast was registered. It can be used to remove the event bcast from the channel.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_update_event_mask**(dragonChannelDescr_t *ch, dragonULInt channel_token, *const* short event_mask)

Update the event mask of an event BCast.

If desired, this function can be called to update the event mask of a BCast object that is stored within a channel. This only updates the triggering event for the given channel.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **channel_token** – is an unsigned integer identifier that was provided when the event bcast was registered.
- **event_mask** – is the new event mask to be applied to the event bcast.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_register_gateways_from_env**()

Register gateway channels from the environment.

When communicating off-node by using a transport agent, the channels library will handle packaging send and receive requests and sending them to the transport agent via gateway channels. To enable this, the serialized descriptors of gateway channels are provided to processes via environment variables. This function should be called by any C code that wishes to commmunicate off-node to other nodes within the allocation served by the Dragon run-time services. Any process started by the Dragon run-time services will have the appropriate gateway channels present in the environment. Calling this function, then registers those gateway channels with the channels library.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_discard_gateways**()

Discard the registered gateway channels.

Any register gateways may be discarded as part of a process' tear down. This is generally not required, but for completeness is available.

-

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_register_gateway**(dragonChannelDescr_t *ch)

Register a channel as a gateway channel.

A local channel can be promoted to be a gateway channel so that messaging through non-local Channels can be passed through and serviced by a transport agent. This operation will take a local channel and register it as a Gateway. A particular channel can only be registered once for a given process or its threads.

This function is not thread-safe in that registering the same channel from multiple threads at the same time could allow a channel to be registered more than once.

Parameters: **ch** – is a pointer to a channel descriptor for the channel to register.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_unregister_gateway**(dragonChannelDescr_t *ch)

Unregister a Channel as a Gateway Channel.

Remove a previously registered Channel from the internally managed list of Gateway Channels. This removes it as a gateway for this process and any associated threads. It does nothing to deregister it for other processes and does nothing to the underlying channel.

Unregistering a gateway channel can only be safely done if all send and receive handles are closed for the current process and its threads, particularly if those send or receive handles refer to a a remote channel. If they are not closed, a send or receive handle could end up referring to a non-existent gateway channel.

Parameters: **ch** – is a pointer to a Channel descriptor for the Channel to deregister.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_gatewaymessage_send_create**(dragonMemoryPoolDescr_t *pool_descr, *const* dragonMessage_t *send_msg, dragonMemoryDescr_t *dest_mem_descr, *const* dragonChannelDescr_t *target_ch, *const* dragonChannelSendAttr_t *send_attr, *const* timespec_t *deadline, dragonGatewayMessage_t *gmsg)

Create a GatewayMessage for sending.

Create a new GatewayMessage for a send operation. The GatewayMessage will be allocated out of the given Managed Memory Pool. After creation, the message can be serialized and sent into a Gateway Channel for processing by a transport agent. The client synchronizes for completion with the transport agent by calling dragon_channel_gatewaymessage_client_send_cmplt().

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **pool_descr** – is a pointer a Managed Memory Pool from which to allocate the message.
- **send_msg** – is a pointer to the message with a payload to send to a remote Channel.
- **dest_mem_descr** – is a pointer to a destination memory descriptor or NULL to let Channels decide where to place the payload on the remote side.
- **target_ch** – is a pointer to a descriptor for the remote target Channel.
- **send_attr** – is a pointer to the send handle attributes associated with the send operation.
- **deadline** – is a pointer to a struct indicating when the actual send operation must finish by as processed by a transport agent. A deadline in the past is equivalent to returning immediately once the operation has been attempted (ie try once without blocking). A NULL values indicates no deadline and dragon_channel_gatewaymessage_transport_send_cmplt() will only return when the operation completes.
- **gmsg** – is a pointer to the Gateway Message to update.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_gatewaymessage_get_create**(dragonMemoryPoolDescr_t *pool_descr, dragonMemoryDescr_t *dest_mem_descr, *const* dragonChannelDescr_t *target_ch, *const* timespec_t *deadline, dragonGatewayMessage_t *gmsg)

Create a GatewayMessage for getting a message.

Create a new GatewayMessage for a get operation. The GatewayMessage will be allocated out of the given Managed Memory Pool. After creation, the message can be serialized and sent into a Gateway Channel for processing by a transport agent. The client synchronizes for completion with the transport agent by calling dragon_channel_gatewaymessage_client_get_cmplt().

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **pool_descr** – is a pointer a Managed Memory Pool from which to allocate the message.

- **dest_mem_descr** – is a descriptor to Managed Memory to place the message into or NULL, which allows the library to decide where to place the message. The final destination of the message is given by the call to dragon_channel_gatewaymessage_client_get_cmplt().
- **target_ch** – is a pointer to a descriptor for the remote target Channel.
- **deadline** – is a pointer to a struct indicating when the actual send operation must finish by as processed by a transport agent. A deadline in the past is equivalent to returning immediately once the operation has been attempted (ie try once without blocking).
- **gmsg** – is a pointer to the Gateway Message to update.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_gatewaymessage_event_create**(dragonMemoryPoolDescr_t *pool_descr, short events, *const* dragonChannelDescr_t *target_ch, *const* timespec_t *deadline, dragonGatewayMessage_t *gmsg)

Create a GatewayMessage for an event.

Create a new GatewayMessage for monitoring for events. The GatewayMessage will be allocated out of the given Managed Memory Pool. After creation, the message can be serialized and sent into a Gateway Channel for processing by a transport agent. The client synchronizes for completion with the transport agent by calling dragon_channel_gatewaymessage_client_event_cmplt().

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **pool_descr** – is a pointer a Managed Memory Pool from which to allocate the message.
- **events** – is a mask of events to monitor for. If any of the requested events occur the operation will complete. The actual event that triggered will be returned in dragon_channel_gatewaymessage_client_event_cmplt().
- **target_ch_ser** – is a pointer to a serialzed descriptor for the remote target Channel.
- **deadline** – is a pointer to a struct indicating when the actual send operation must finish by as processed by a transport agent. A deadline in the past is equivalent to returning immediately once the operation has been attempted (ie try once without blocking).
- **gmsg** – is a pointer to the Gateway Message to update.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_gatewaymessage_destroy**(dragonGatewayMessage_t *gmsg)

Destroy a GatewayMessage.

Destroy a GatewayMessage of any operation type. This call exists to allow a process to cleanup a GatewayMessage that is not otherwise needed and its use by the completion calls. Typically a direct call to this function is not needed.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters: **gmsg** – is a pointer to the Gateway Message to destroy.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_gatewaymessage_serialize**(*const* dragonGatewayMessage_t *gmsg, dragonGatewayMessageSerial_t *gmsg_ser)

Serialize a GatewayMessage.

Serialize a GatewayMessage so that another process can interact with the GatewayMessage once the serialized representation is attached to. The serialized representation can be communicated with another process (the transport agent) through any means (typically a Gateway Channel).

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **gmsg** – is a pointer to the Gateway Message to serialize.
- **gmsg_ser** – is a pointer to the serialized Gateway Message message structure to update.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_gatewaymessage_serial_free**(dragonGatewayMessageSerial_t *gmsg_ser)

Free a serialize GatewayMessage.

Clean up a serialized representation of a Gateway Message.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters: **gmsg_ser** – is a pointer to the serialized Gateway Message message structure to clean up.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_gatewaymessage_attach**(*const* dragonGatewayMessageSerial_t *gmsg_ser, dragonGatewayMessage_t *gmsg)

Attach to a serialized GatewayMessage.

Attach to an existing Gateway Message from a serialized representation. Once attached, a process can interact with the Gateway Message and access its members.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **gmsg_ser** – is a pointer to the serialized Gateway Message message structure to clean up.
- **gmsg** – is a pointer to the Gateway Message to update.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_gatewaymessage_detach**(dragonGatewayMessage_t *gmsg)

Detach from a GatewayMessage.

Detaching from a Gateway Message is not typically done directly. The completion functions will internally do this and use this function.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters: **gmsg** – is a pointer to the Gateway Message to detach from.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_gatewaymessage_transport_send_cmplt**(dragonGatewayMessage_t *gmsg, *const* dragonError_t op_err)

Complete a send operation from a transport agent.

Once a transport agent has completed a send operation for the Gateway Message, this call is used to coordinate completion with the requesting client. This function will manage cleanup of the Gateway Message. The given Gateway Message is not usable after return from this call.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **gmsg** – is a pointer to the Gateway Message for send to complete.
- **op_err** – is an error code to propagate back to the client.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_gatewaymessage_client_send_cmplt**(dragonGatewayMessage_t *gmsg, *const* dragonWaitMode_t wait_mode)

Complete a send operation from a client.

Wait for completion of the given Gateway Message send operation. This function will manage cleanup of the Gateway Message. The given Gateway Message is not usable after return from this call.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters: **gmsg** – is a pointer to the Gateway Message for send to complete.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_gatewaymessage_transport_get_cmplt**(dragonGatewayMessage_t *gmsg, dragonMessage_t *msg_recv, *const* dragonError_t op_err)

Complete a get operation from a transport agent.

Once a transport agent has completed a get operation for the Gateway Message, this call is used to coordinate completion with the requesting client. This function will manage cleanup of the Gateway Message. The given Gateway Message is not usable after return from this call.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **gmsg** – is a pointer to the Gateway Message for get to complete.

- **msg_recv** – is a pointer to the received Message that will be provided back to the client when the op_err argument is set to DRAGON_SUCCESS. Otherwise, it should be NULL.
- **op_err** – is an error code to propagate back to the client. DRAGON_SUCCESS indicates the requested message is returned. Otherwise, the error code reflects the error that occurred.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred on the call.

---

dragonError_t **dragon_channel_gatewaymessage_client_get_cmplt**(dragonGatewayMessage_t *gmsg, dragonMessage_t *msg_recv, *const* dragonWaitMode_t wait_mode)

Complete a get operation from a client.

Wait for completion of the given Gateway Message get operation. This function will manage cleanup of the Gateway Message. The given Gateway Message is not usable after return from this call.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **gmsg** – is a pointer to the Gateway Message for get to complete.
- **msg_recv** – is a pointer to a Message that will be updated with the received message when the return code is DRAGON_SUCCESS. Otherwise, the message is left uni

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_gatewaymessage_transport_event_cmplt**(dragonGatewayMessage_t *gmsg, *const* dragonULInt event_result, *const* dragonError_t op_err)

Complete an event operation from a transport agent.

Once a transport agent has completed an event operation for the Gateway Message, this call is used to coordinate completion with the requesting client. This function will manage cleanup of the Gateway Message. The given Gateway Message is not usable after return from this call.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **gmsg** – is a pointer to the Gateway Message for event to complete.
- **result** – is the result of calling the poll operation if there was a result to return. A valid result is only for those poll operations that return a result when the return value is DRAGON_SUCCESS.
- **op_err** – is an error code to propagate back to the client.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_gatewaymessage_client_event_cmplt**(dragonGatewayMessage_t *gmsg, dragonULInt *event, *const* dragonWaitMode_t wait_mode)

Complete an event operation from a client.

Wait for completion of the given Gateway Message event operation. This function will manage cleanup of the Gateway Message. The given Gateway Message is not usable after return from this call.

This function is not intended for most Channels users. It is used internally to support Channels operations on remote nodes and other purpose-built libraries.

Parameters:
- **gmsg** – is a pointer to the Gateway Message for send to complete.
- **event** – is a pointer that will be updated with the triggered event.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

Variables

---

static dragonMemoryDescr_t **const* **DRAGON_CHANNEL_SEND_TRANSFER_OWNERSHIP** = (dragonMemoryDescr_t*)1

Constant to be used for transfer of ownership of the message payload area on a send operation.

---

static timespec_t *const* **DRAGON_CHANNEL_BLOCKING_NOTIMEOUT** = {INT32_MAX, INT32_MAX}

Constant to be used for no timeout.

For send_msg and get_msg this would mean to wait with no timeout.

---

static timespec_t *const* **DRAGON_CHANNEL_TRYONCE_TIMEOUT** = {0, 0}

Try Once Timeout.

This indicates a try-once attempt. This works both on-node and off-node.

struct `dragonChannelAttr_t`

#include <channels.h>
The attributes structure for tuning a Message.

This structure contains members that can tune Message behavior and can be used by a transport agent for operation handling for operations on remote Channels.

Public Members

dragonC_UID_t **`c_uid`**

size_t **`bytes_per_msg_block`**

size_t **`capacity`**

dragonLockKind_t **`lock_type`**

dragonChannelOFlag_t **`oflag`**

dragonChannelFC_t **`fc_type`**

dragonULInt **`flags`**

dragonMemoryPoolDescr_t ***`buffer_pool`**

size_t **`max_spinners`**

size_t **`max_event_bcasts`**

int **`blocked_receivers`**

int **`blocked_senders`**

size_t **`num_msgs`**

size_t **`num_avail_blocks`**

bool **`broken_barrier`**

int **`barrier_count`**

struct `dragonChannelSendAttr_t`

#include <channels.h>
Send Handle Attributes.

These attributes are provided on send handle creation.

Public Members

dragonUUID **`sendhid`**

Used internally by the Dragon run-time services.

dragonChannelSendReturnWhen_t **`return_mode`**

When to return from a send.

timespec_t **`default_timeout`**

Default timeout used when NULL is provided as the timeout override.

dragonWaitMode_t **wait_mode**

Either IDLE wait or SPIN wait may be specified.

struct dragonChannelRecvAttr_t

#include <channels.h>
Receive Handle Attributes.

The attributes are provided on receive handle creation.

Public Members

dragonChannelRecvNotif_t **default_notif_type**

Notification type is currently unused.

timespec_t **default_timeout**

Default timeout used when NULL is provided as the timeout override.

int **signal**

Signal is currently unused.

dragonWaitMode_t **wait_mode**

Either IDLE wait or SPIN wait may be specified.

struct dragonChannelDescr_t

#include <channels.h>
An opaque channel descriptor.

When a channel created, a channel descriptor is initialized for the current process. These channel descriptors may be shared with other processes by first serializing them, and then passing the serialized descriptor to another process. The other process must then attach to the channel using the serialized descriptor. Attaching and creating are the two means of initializing a channel descriptor.

struct dragonChannelSerial_t

#include <channels.h>
A serialized channel descriptor.

It should be treated as binary data with the given length.

Public Members

size_t **len**

The length of the serialized descriptor in bytes.

uint8_t ***data**

struct dragonChannelSendh_t

#include <channels.h>
A Channel Send Handle.

A send handle must be initialized and opened prior to sending data. All fields internal use only.

struct dragonChannelRecvh_t

#include <channels.h>
A Channel Receive Handle.

A receive handle must be initialized and opened prior to receiving data. All fields internal use only.

struct dragonMessageAttr_t

#include <channels.h>
The attributes structure for tuning a Message.

This structure contains members that can tune Message behavior and can be used by a transport agent for operation handling for operations on remote Channels.

Public Members

dragonULInt **hints**

Placeholder of future hints about this message

dragonULInt **clientid**

An identifier of the process that sent this message

dragonUUID **sendhid**

An identifier of the send handle for the sending process used for ordering

struct dragonMessage_t

#include <channels.h>
A Message structure.

This structure has no public members and is used in send and get operations.

struct dragonChannelEventNotification_t

#include <channels.h>
An Event Notification structure.

This is the event that occurred in a channelset notification.

Public Members

int **user_token**

An identifier associated with this event when it was registered.

short **revent**

The event that occurred.

struct dragonGatewayMessageHeader_t

#include <channels.h>
The Gateway Message Header.

This is provided here but used internally in the channels implementation and by a transport service when reading gateway messages from a gateway channel. All fields are internal use only unless defininining a new transport service beyond those provided with Dragon. All fields are set via gateway message creation.

Public Members

dragonGatewayMessageKind_t ***msg_kind**

A send, get, or event message

dragonULInt ***target_hostid**

Hostid identifying the target of this request.

dragonULInt ***has_deadline**

If there is a timeout, then this points to true.

dragonULInt ***deadline_sec**

Seconds of the timeout

dragonULInt ***deadline_nsec**

Nanoseconds part of timeout

atomic_int_fast64_t ***client_cmplt**

Set to 1 when client has completed pickup.

dragonULInt ***cmplt_bcast_offset**

Offset of the completion bcast

dragonULInt ***target_ch_ser_offset**

The serialized descriptor of the target channel

dragonULInt ***target_ch_ser_nbytes**

Number of bytes in target channel serialized descriptor.

dragonULInt ***send_payload_cleanup_required**

Whether the transport is required to clean up.

dragonULInt ***send_payload_buffered**

If points to true, then send_payload_offset is offset to serialized descriptor

dragonULInt ***send_payload_offset**

When sending, the location of the message to send

dragonULInt ***send_payload_nbytes**

Number of bytes in send payload.

dragonULInt ***send_clientid**

used internally for message ordering.

dragonULInt ***send_hints**

provided on send operation and passed along.

dragonULInt ***send_return_mode**

provided on send operation and passed along.

dragonULInt ***has_dest_mem_descr**

On a get, a destination for the received message may be specified.

dragonULInt ***dest_mem_descr_ser_offset**

Where to put it if a destination was provided.

dragonULInt ***dest_mem_descr_ser_nbytes**

Size of descriptor when told where to place received message.

dragonULInt ***op_rc**

The gateway operation return code

dragonULInt ***event_mask**

On poll this is the poll event to monitor. On poll response it is the event that occurred.

dragonUUID ***sendhid**

Used in send ordering.

struct **dragonGatewayMessage_t**

#include <channels.h>
The Gateway Message structure for interacting with a transport agent.

This structure groups together all information about a message needed by a transport agent for completing the operation. Public members can be directly accessed for fast access to relevant data about the operation while private members facilitate coordination with a client process.

Public Members

dragonChannelSerial_t **target_ch_ser**

Serialize descriptor of the target Channel

dragonGatewayMessageKind_t **msg_kind**

The kind of message (send, get, event)

dragonULInt **target_hostid**

The hostid the target Channel is on

timespec_t **deadline**

When the operation must be completed by

dragonChannelSendReturnWhen_t **send_return_mode**

When a send operation should return to the caller

dragonMessage_t **send_payload_message**

The message being sent for send operations

dragonMemorySerial_t ***send_dest_mem_descr_ser**

Optional destination serialized memory descriptor for sends

dragonMemorySerial_t ***get_dest_mem_descr_ser**

Optional destination serialized memory descriptor for gets

short **event_mask**

Mask of events to monitor for event operations

struct dragonGatewayMessageSerial_t

#include <channels.h>
A serialized Gateway Message handle.

This structure cannot be used directly to manipulate a message. All manipulation of the Gateway Message occurs via its API functions. A process gets a serialized Gateway Message by calling serialize on a valid Gateway Message object.

The data is a serialized representation of a Gateway Message object that can be passed between processes and used to attach to the same object in another process or thread, such as a transport agent.

Public Members

size_t **len**

The length in bytes of the data member

uint8_t ***data**

A pointer to buffer containing the serialized representation

Functions

dragonError_t dragon_channel_send_attr_init(dragonChannelSendAttr_t *send_attr)

Initialize a send attributes structure.

When custom send attributes are desired, this function should be called first, to initialize the to default values. Then the user may override desired attributes before using it in creating a send handle.

Parameters: **send_attr** – is a pointer to the send attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_recv_attr_init**(dragonChannelRecvAttr_t *recv_attr)

Initialize a receive attributes structure.

When custom receive attributes are desired, this function should be called first to initialize them to default values. Then the user may override desired attributes before using it in creating a receive handle.

Parameters: **recv_attr** – is a pointer to the receive attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_attr_init**(dragonChannelAttr_t *attr)

Initialize a channel attributes structure.

When custom channel attributes are desired, this function should be called first to initialize them to default values. Then the user may override desired attributes before using it in creating a channel.

Parameters: **attr** – is a pointer to the channel attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_send_attr_destroy**(dragonChannelSendAttr_t *send_attr)

Destroy a channel send handle attributes structure.

Release any allocations or resources associated with a send handle attributes structure. This does not destroy the underlying channel.

Parameters: **send_attr** – is a pointer to the channel send handle attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_recv_attr_destroy**(dragonChannelRecvAttr_t *recv_attr)

Destroy a channel recv handle attributes structure.

Release any allocations or resources associated with a recv handle attributes structure. This does not destroy the underlying channel.

Parameters: **recv_attr** – is a pointer to the channel send handle attributes structure.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_attr_destroy**(dragonChannelAttr_t *attr)

Destroy a channel attributes structure.

Release any allocations in the attributes structure. This does not destroy the underlying channel.

Parameters: **attr** – is a pointer to the attributes structure that was previously inited.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_create**(dragonChannelDescr_t *ch, *const* dragonC_UID_t c_uid, dragonMemoryPoolDescr_t *pool_descr, *const* dragonChannelAttr_t *attr)

Create a channel in a memory pool.

Create a channel in a memory pool with the given c_uid. While the c_uid is not enforced to be unique across the Dragon run-time services by this api, that is desirable and is left to the user of this API call. Unique c_uid values are enforced by this API call at the process level. If you need globally unique c_uids, then use the managed channel API instead which coordinates with the Dragon run-time services to insure global uniqueness. For transparent multi-node communication between channels, c_uids do not have to be unique across all nodes.

Parameters:
- **ch** – is the channel descriptor which will be initialzed from this call.
- **c_uid** – is a channel identifier. It must be unique on a process level and should be unique across all nodes. Read about uniqueness in the general description.
- **pool_descr** – is the pool in which to allocate this channel.
- **attr** – are the attributes to be used in creating this channel. If providing attributes, make sure you call dragon_channel_attr_init first. Otherwise, NULL can be provided to get default attributes.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_destroy**(dragonChannelDescr_t *ch)

Destroy a channel.

Destroying a channel can only be done on the node where the channel is located. Destroying a channel frees the allocated memory in the memory pool and invalidates any use of the channel from this or other processes.

Parameters: **ch** – is the channel descriptor to destroy.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_get_pool**(*const* dragonChannelDescr_t *ch, dragonMemoryPoolDescr_t *pool_descr)

Get pool from channel.

Given a channel, get the pool from the channel where it resides.

Parameters:
- **ch** – is a pointer to the channel descriptor.
- **pool_descr** – is a pointer the the pool descriptor that will be initialized by this call.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_serialize**(*const* dragonChannelDescr_t *ch, dragonChannelSerial_t *ch_ser)

Serialize a channel to be shared with another process.

Parameters:
- **ch** – is a channel descriptor for a channel.
- **ch_ser** – is a serialized channel descriptor that may be shared with another process.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_serial_free**(dragonChannelSerial_t *ch_ser)

Free the internal resources of a serialized channel descriptor.

This frees internal structures of a serialized memory descriptor. It does not destroy the channel itself.

Parameters: **ch_ser** – is a serialized channel descriptor.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_attach**(*const* dragonChannelSerial_t *ch_ser, dragonChannelDescr_t *ch)

Attach to a channel.

Calling this attaches to a channel by using a serialized channel descriptor that was passed to this process. The serialized channel descriptor must have been created using the dragon_channel_serialize function.

Parameters:
- **ch_ser** – is a pointer to the serialized channel descriptor.
- **ch** – is a pointer to a channel descriptor that will be initialized by this call.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_detach**(dragonChannelDescr_t *ch)

Detach from a channel.

Calling this will clean up any local references to a channel and release any local resources needed for communicating with the channel. It does not destroy the channel itself.

Parameters: **ch** – is the channel descriptor from which to detach.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_get_hostid**(*const* dragonChannelDescr_t *ch, dragonULInt *hostid)

Get the host identifier for a channel.

The host identifier is used to decide on which node the channel is hosted. This determines if it is a local channel or a remote channel. Sends and receives to/from remote channels are handled through gateway channels that must be registered prior to sending and or receiving. Gateway channels are shared with a process via an environment variable or variables. See register_gateways_from_env for more information on registering gateway channels. Host ids are arbitrary unsigned long integers and cannot not be interpreted or inferred from any other data.

Parameters:
- **ch** – is an initialized channel descriptor
- **hostid** – is a pointer to space to provide the hostid.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_get_uid_type**(*const* dragonChannelSerial_t *ch_ser, dragonULInt *cuid, dragonULInt *type)

Get the channel's cuid and/or type.

From a serialized channel descriptor, this function will return the cuid of the channel and the type of the channel. The channel is not attached while doing this.

Parameters:
- **ch_ser** – is a pointer to a serialied channel descriptor.
- **cuid** – is a pointer to a location there the cuid will be stored. If NULL is provided, the cuid is not copied from the channel descriptor.
- **type** – is a pointer to a location where the channel's type will be stored. If NULL is provided, the type is not copied from the channel descriptor.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_pool_get_uid_fname**(*const* dragonChannelSerial_t *ch_ser, dragonULInt *muid, char **pool_fname)

Get the muid and filename of a channel's pool.

From a serialized channel descriptor, this function returns information about the pool the channel is allocated in, including the muid of the pool and the pool's filename. The channel is not attached while doing this.

Parameters:
- **ch_ser** – is a pointer to a serialized channel descriptor.
- **muid** – is a pointer to a location where the muid of the pool will be stored. If NULL is provided, then the muid is not copied from the channel.
- **pool_fname** – is the filename associated with the shared memory of the channel's pool.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

bool **dragon_channel_is_local**(*const* dragonChannelDescr_t *ch)

Check to see is a channel is local or not.

Returns true if a channel is local and false if not.

Parameters: **ch** – is an intialized channel descriptor.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_sendh**(*const* dragonChannelDescr_t *ch, dragonChannelSendh_t *ch_sh, *const* dragonChannelSendAttr_t *attr)

Initialize a send handle on a channel.

Calling this initializes a send handle on a channel. To use the send handle it must also be opened using the dragon_chsend_open function. Messages sent with a single send handle are guaranteed to be received from the channel in the same order. Order is maintained by both send and receive handles. If customized send handle attributes are to be supplied, dragon_channel_send_attr_init should be called first to initialize all attributes to default values before customizing the desired values.

Parameters:
- **ch** – is an initialized channel descriptor.
- **ch_sh** – is a pointer to a send handle descriptor that will be initialized by this call.
- **attr** – are send handle attributes that may be supplied when initializing the send handle. Providing NULL will result in the default send handle attributes being applied.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chsend_get_attr**(*const* dragonChannelSendh_t *ch_sh, dragonChannelSendAttr_t *attr)

Get the attributes of a send handle.

Copy the attributes from a send handle into a send attributes structure. The handle does not need to be open in order to get the attributes.

Parameters:
- **ch_sh** – A pointer to the channel send handle.
- **attr** – A pointer to the channel send handle attributes structure to copy into.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chsend_open**(dragonChannelSendh_t *ch_sh)

Open a channel send handle.

Open a send handle on a channel. Once opened, the send handle can be used for sending messages.

Parameters: **ch_sh** – is a pointer to an initialized send handle.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chsend_close**(dragonChannelSendh_t *ch_sh)

Close a send handle.

Once sending is done on a particular send handle, it should be closed.

Parameters: **ch_sh** – is a pointer to an open send handle.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_recvh**(*const* dragonChannelDescr_t *ch, dragonChannelRecvh_t *ch_rh, *const* dragonChannelRecvAttr_t *rattrs)

Initialized a channel receive handle.

Initialize a receive handle structure given a channel. To use the receive handle it must also be opened using the dragon_chrecv_open function. Messages received with a single receive handle are guaranteed to be received from the channel in the same order they were sent. Order is maintained by both send and receive handles. If customized receive handle attributes are to be supplied, dragon_channel_recv_attr_init should be called first to initialize all attributes to default values before customizing the desired values.

Parameters:
- **ch** – is an initialized channel descriptor
- **ch_rh** – is a pointer to a structure that will be initalized by this call.
- **rattrs** – is a pointer to receive handle attributes. If NULL is provided, default receive handle attributes will be used.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chrecv_get_attr**(*const* dragonChannelRecvh_t *ch_rh, dragonChannelRecvAttr_t *attr)

Get the attributes for a receive handle.

Copy the attributes from a receive handle into a receive attributes structure. The handle does not need to be open in order to get the attributes.

Parameters:
- **ch_rh** – A pointer to the channel receive handle.
- **attr** – A pointer to the channel receive handle attributes structure to copy into.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chrecv_open**(dragonChannelRecvh_t *ch_rh)

Open a channel receive handle.

Once opened, a receive handle can be used to receive messages from a channel. Messages received from a receive handle are guaranteed to be received in the order they were sent.

Parameters: **ch_rh** – is the receive handle to be opened.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chrecv_close**(dragonChannelRecvh_t *ch_rh)

Close a channel receive handle.

Once a receive handle is no longer to be used for receiving, the handle should be closed.

Parameters: **ch_rh** – is a pointer to an open channel receive handle.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_chsend_send_msg**(*const* dragonChannelSendh_t *ch_sh, *const* dragonMessage_t *msg_send, dragonMemoryDescr_t *dest_mem_descr, *const* timespec_t *timeout_override)

Send a message into a channel.

Much like putting data in a queue, messages can be sent through channels. Sending a message into a channel can be done in several ways. The destination memory descriptor (i.e. dest_mem_descr) may be NULL in which case the memory associated with the message to send will be copied upon sending. In that way the caller retains ownwership of the messages original memory allocation. If dest_mem_descr is non-NULL, then the message is copied into the dest_mem_descr before sending it. However, if dest_mem_descr is the special value DRAGON_CHANNEL_SEND_TRANSFER_OWNERSHIP, then this indicates the caller wishes to transfer the ownership of the memory allocation in msg_send to the receiver of this message. In this way, a zero-copy send operation is possible when sending on-node between processes. Using transfer of ownership also has performance advantages when communicating between nodes since the path through the transport agent then has fewer copies as well.

If a NULL dest_mem_descr is provided, the memory allocation for the message copy is made from either the channel's pool (if the channel is local) or from the default memory pool for the node where the message is being sent from.

When a memory allocation is required because the dest_mem_descr is NULL, calling this function may block the sender until the memory allocation is available. This call will also block while waiting for exclusive access to the channel. If a timeout occurs on sending, then it was because the required memory was not available.

Parameters:
- **ch_sh** – is a pointer to an initialized and open send handle.
- **msg_send** – is a pointer to a valid channel message descriptor.
- **dest_mem_descr** – is a pointer to a destination memory descriptor. The detailed description above has more details on the valid values for this argument.
- **timeout_override** – is a pointer to a timeout structure that may be used to override the default send timeout as provided in the send handle attributes. A timeout of zero seconds and zero nanoseconds will result in a try-once call of send. A value of NULL will result in using the default timeout from the send handle attributes.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chrecv_get_msg**(*const* dragonChannelRecvh_t *ch_rh, dragonMessage_t *msg_recv)

Receive a message from a channel.

Calling this receives the next available message from a channel without blocking.

Parameters:
- **ch_rh** – is a pointer to an intialized and open receive handle structure.
- **msg_recv** – is a pointer to a message structure that will be initialized with the received message.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred. If no message is immediately available it will return DRAGON_CHANNEL_EMPTY.

---

dragonError_t **dragon_chrecv_get_msg_notify**(dragonChannelRecvh_t *ch_rh, dragonMessage_t *msg_recv, dragonBCastDescr_t *sync)

Asynchronously receive a message from a channel.

This function is not currently implemented.

Parameters:
- **ch_rh** – is a pointer to an initialized, open receive handle.
- **msg_recv** – is a pointer to a message descriptor to be initialzed aynchronously.
- **sync** – is a pointer to a BCast descriptor which points to a valid BCast object. The calling process can then wait on this BCast for a message to receive. This can be safely done by a thread of the calling process or by the process itself.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_chrecv_get_msg_blocking**(*const* dragonChannelRecvh_t *ch_rh, dragonMessage_t *msg_recv, *const* timespec_t *timeout_override)

Receive a message from a channel.

This receives a message from a channel into the supplied message structure. A timeout override can be supplied to override the default timeout found in the channel receive handle. If the timeout_override is NULL, the default receive handle timeout is used. A timeout of zero seconds and zero nanoseconds will result in try once attempt which will return DRAGON_CHANNEL_EMPTY if no message is available.

Parameters:
- **ch_rh** – is a pointer to an initialized, open receive handle.
- **msg_recv** – is a pointer to a message descriptor which will be initialized by this call upon successful completion should a message be received.
- **timeout_override** – is a pointer to a structure that contains the timeout to use in place of the default handle timeout. If NULL, the default handle timeout is used. If zero seconds and nanoseconds, the call is non-blocking and returns the next message if availble or DRAGON_CHANNEL_EMPTY if no message is currently available.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_poll**(*const* dragonChannelDescr_t *ch, dragonWaitMode_t wait_mode, *const* short event_mask, *const* timespec_t *timeout, dragonULInt *result)

Poll a channel for status or the occurrence of an event.

Polling may be done on any of the poll values found on dragonChannelEvent_t. The polling values POLLIN and POLLOUT may be done together with the value DRAGON_CHANNEL_POLLINOUT. All other polling event_masks must be done separately. They are not supported in combination with each other.

Parameters:
- **ch** – A channel descriptor for a channel either on-node or off-node.
- **wait_mode** – A choice between IDLE waiting or SPIN waiting for events polling. It is only relevant when polling for events.
- **event_mask** – This specifies one of the dragonChannelEvent_t constants.

- **timeout** – NULL indicates blocking with no timeout. Otherwise, blocks for the specified amount of time. If (0,0) is provided, then it is a non-blocking call.
- **result** – For all but the DRAGON_CHANNEL_POLLSIZE the result will be a 64-bit cast of the event_mask field. With DRAGON_CHANNEL_POLLSIZE it is the number of messages currently in the channel.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_message_count**(*const* dragonChannelDescr_t *ch, uint64_t *count)

Get the number of messages in a channel.

This will return the number of messages found in a channel. The number of messages will be correct when queried, but may change before this function returns to the caller if other processes are interacting with the channel by sending or receiving messages.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **count** – is a pointer to space for a 64 bit unsigned integer that will hold the number of messages found in the channel.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_get_attr**(*const* dragonChannelDescr_t *ch, dragonChannelAttr_t *attr)

Get the attributes of a channel.

Calling this will initialize a channel attributes structure with read-only and user-specified attributes of the channel. Read-only attributes are supplied on this call so a caller may inspect the current state of the channel.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **attr** – is a pointer to a channel attributes structure that will be initialized by this call.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_register_gateways_from_env**()

Register gateway channels from the environment.

When communicating off-node by using a transport agent, the channels library will handle packaging send and receive requests and sending them to the transport agent via gateway channels. To enable this, the serialized descriptors of gateway channels are provided to processes via environment variables. This function should be called by any C code that wishes to commmunicate off-node to other nodes within the allocation served by the Dragon run-time services. Any process started by the Dragon run-time services will have the appropriate gateway channels present in the environment. Calling this function, then registers those gateway channels with the channels library.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_discard_gateways**()

Discard the registered gateway channels.

Any register gateways may be discarded as part of a process' tear down. This is generally not required, but for completeness is available.

-

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_register_gateway**(dragonChannelDescr_t *ch)

Register a channel as a gateway channel.

A local channel can be promoted to be a gateway channel so that messaging through non-local Channels can be passed through and serviced by a transport agent. This operation will take a local channel and register it as a Gateway. A particular channel can only be registered once for a given process or its threads.

This function is not thread-safe in that registering the same channel from multiple threads at the same time could allow a channel to be registered more than once.

Parameters: **ch** – is a pointer to a channel descriptor for the channel to register.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

dragonError_t **dragon_channel_unregister_gateway**(dragonChannelDescr_t *ch)

Unregister a Channel as a Gateway Channel.

Remove a previously registered Channel from the internally managed list of Gateway Channels. This removes it as a gateway for this process and any associated threads. It does nothing to deregister it for other processes and does nothing to the underlying channel.

Unregistering a gateway channel can only be safely done if all send and receive handles are closed for the current process and its threads, particularly if those send or receive handles refer to a a remote channel. If they are not closed, a send or receive handle could end up referring to a non-existent gateway channel.

Parameters: **ch** – is a pointer to a Channel descriptor for the Channel to deregister.
Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_add_event_bcast**(dragonChannelDescr_t *ch, dragonBCastSerial_t *ser_bcast, *const* short event_mask, int user_token, dragonULInt *channel_token)

Create a channel set from a list of channels with possible attributes.

You create a channel set when you wish to poll across a set of channels. The details of how this is accomplished is handled by the channel set API. In this call you provide a list of the channels you wish to poll across. To get the most efficient implementation of this multi-channel poll, if it is in your control, specify the same default bcast event descriptor in the channel attributes for each channel when it is created.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **ser_bcast** – is a serialized BCast decriptor to be added to the channel.
- **event_mask** – is the event mask to be monitored in the channel and signaled via the given BCast object.
- **user_token** – is a user-supplied token to be provided when the BCast is triggered.
- **channel_token** – is a pointer to space for an unsigned integer that is returned as a channel identifier when the event is triggered for this channel. This is used by the channel itself, not by the caller, but is provided for subsequent event bcast calls on this channel.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_remove_event_bcast**(dragonChannelDescr_t *ch, dragonULInt channel_token)

Remove an event BCast from this channel.

The channel_token returned when a process adds an event bcast to a channel can be used to subsequently delete the even bcast from the channel.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **channel_token** – is the channel token that was provided when the event bcast was registered. It can be used to remove the event bcast from the channel.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

---

dragonError_t **dragon_channel_update_event_mask**(dragonChannelDescr_t *ch, dragonULInt channel_token, *const* short event_mask)

Update the event mask of an event BCast.

If desired, this function can be called to update the event mask of a BCast object that is stored within a channel. This only updates the triggering event for the given channel.

Parameters:
- **ch** – is a pointer to an initialized channel descriptor.
- **channel_token** – is an unsigned integer identifier that was provided when the event bcast was registered.
- **event_mask** – is the new event mask to be applied to the event bcast.

Returns: DRAGON_SUCCESS or a return code to indicate what problem occurred.

Variables

---

static dragonMap_t ***dg_channels** = NULL

---

static dragonList_t ***dg_gateways** = NULL

Example

The main purpose of this example is to demonstrate how you can program directly to the channels API. As such, much of the work is done at the Dragon core channels API level. However, there is some bootstrap code that is written in Python to get everything started. The Python program bootstraps a C example of using the Channels API. The Python code given here shows how the program is started in the usage function in Listing 13. A simple run is given in Listing 12. The program starts up a specifiifed number of ring processes, potentially on different nodes of a multi-node allocation. Each ring process runs a specified number of iterations of passing a message around the ring. The final ring process receives a message from the previous ring process and forwards it back to the beginning of the ring to be sent around atain. The program reports the average time it takes to forward the message from once ring process to the next in the ring.

Listing 12 A Sample Run of the Ring Demo

```
(_env) root ➜ .../hpc-pe-dragon-dragon/examples/dragon_core/ring $ dragon ring.py 2 100
Ring proc exited...
Ring proc exited...
Test Passed.
The average time per message transfer was 12.092739925719798 microseconds.
Main proc exiting...
+++ head proc exited, code 0
(_env) root ➜ .../hpc-pe-dragon-dragon/examples/dragon_core/ring $
```

The bootstrap code shown in Listing 13 demonstrates how a process can be started on each node of an allocation or cluster. The default placement strategy of round-robin means that each process is started on a different node. The *start_ringproc* function then uses subprocess.Popen to start a second process on the node. The standard output from *ringproc* is the serialized descriptor of a channel that the *ringproc* instance will send messages to. That serialized descriptor is fed back to the main program's process to be provided as the receive channel for the next process in the ring of processes.

The design of this bootstrap program allows for the program to be started with as many processes and iterations as desired. So all or some subset of nodes may be used from a cluster or allocation. Or you can start more processes than nodes that are available in the cluster/allocation and the ring will simply overlap some nodes using the round-robin placement of processes.

The bootstrap application sends the message around the ring the number of *iteration* times and it times that total time and computes the average time it takes for a message transfer between channels. Note that the send channel for each *ringproc* co-exists on the same node as the *ringproc* process instance. So not only are the *ringprocs* distributed across nodes, but their send channels for the ring have a similar distribution across nodes.

Listing 13 Ring Demo Bootstrap Code

```python
import dragon
import multiprocessing as mp
import subprocess
import sys
import dragon.channels as dch
import dragon.managed_memory as dm
import dragon.infrastructure.parameters as dp
import dragon.infrastructure.facts as df
import dragon.utils as du
import time

def start_ringproc(iterations, cuid, receive_from_channel_sdesc, ret_queue):
    proc = subprocess.Popen(['/workspaces/hpc-pe-dragon-dragon/examples/dragon_core/ring/ringproc', str(iterations), str(cuid), receive_from_channel_sdesc], stdout=subprocess.PIPE)
    send_to_channel_sdesc = proc.stdout.readline()
    while len(send_to_channel_sdesc.strip()) == 0:
        send_to_channel_sdesc = proc.stdout.readline()
    ret_queue.put(send_to_channel_sdesc)
    proc.wait()
    if proc.returncode != 0:
        print('*******Proc exited with rc=', proc.returncode, flush=True)

def usage():
    print('usage: dragon ring.py <num_procs> <iterations>')
    print('    <num_procs> is the number of processes to start, one per node.')
    print('    <iterations> is the number of times each process forwards a message')
    print('                to the next node.')
    print('    The program creates a ring across the user specified number of')
    print('    nodes and sends a message around a ring of nodes. The num_procs')
    print('    and iterations must be greater than 0.')
    sys.exit(1)

def main():
    try:
        if len(sys.argv) != 3:
            raise ValueError()

        mp.set_start_method('dragon')
        ring_size = int(sys.argv[1])
        iterations = int(sys.argv[2])
        if iterations <= 0 or ring_size <= 0:
            raise ValueError()
    except:
        usage()

    pool = dm.MemoryPool.attach(du.B64.str_to_bytes(dp.this_process.default_pd))
    origin_channel = dch.Channel(pool, df.BASE_USER_MANAGED_CUID)
    receive_sdesc = du.B64.bytes_to_str(origin_channel.serialize())
    final_channel = dch.Channel(pool, df.BASE_USER_MANAGED_CUID+1)
    final_sdesc = du.B64.bytes_to_str(final_channel.serialize())
    origin_send_sdesc = receive_sdesc

    ret_queue = mp.Queue()
    mp_procs = []
    for i in range(1,ring_size):
        proc = mp.Process(target=start_ringproc, args=(str(iterations), str(i+df.BASE_USER_MANAGED_CUID+1), receive_sdesc, ret_queue))
        proc.start()
        mp_procs.append(proc)
        receive_sdesc = ret_queue.get().strip()

    # This final process starts on the current node and completes the ring. It
    # also provides the destination for the final message to be returned.
    proc = subprocess.Popen(['/workspaces/hpc-pe-dragon-dragon/examples/dragon_core/ring/ringproc', str(iterations), str(df.BASE_USER_MANAGED_CUID), receive_sdesc, origin_send_sdesc, final_sdesc], stdout=subprocess.PIPE)

    reader = dch.ChannelRecvH(final_channel)
    writer = dch.ChannelSendH(origin_channel)
    reader.open()
    writer.open()
    start = time.perf_counter()
    writer.send_bytes(b'hello', timeout=None, blocking=True)
    last_msg = reader.recv_bytes(timeout=None, blocking=True)
    stop = time.perf_counter()

    avg_time = (stop - start) / (iterations*ring_size)
    proc.wait()
    print('Ring proc exited...', flush=True)
    for proc in mp_procs:
        proc.join()
        print('Ring proc exited...', flush=True)
    if last_msg == b'hello':
        print('Test Passed.', flush=True)
        print(f'The average time per message transfer was {avg_time*1e6} microseconds.')
    else:
        print('Test Failed.', flush=True)
```

```
    print('Main proc exiting...', flush=True)


if __name__ == '__main__':
    main()
```

The code in Listing 14 is the C program that uses the Channels API to receive and send a message. There is one process running this code on each node of the ring. The code takes three or five arguments. The three argument case is used for all but the last process in the ring. The code is given a receive channel descriptor where it will receive a message from in the ring. It then creates a new channel where it will send the message to. The send channel descriptor is written to standard output which is monitored to read it and then provide that channel descriptor to the next instance of the *ringproc* code from which it receives its message.

Comments in the code describe why each API call is made. The pattern used here checks return codes from all calls and prints to standard error should there be any errors. Since standard error is captured by Dragon, any error messages are displayed back to the user.

Listing 14 Ring Demo Process Code

```c
#include <dragon/channels.h>
#include <dragon/return_codes.h>
#include <dragon/utils.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {

    if (argc < 4) {
        fprintf(stderr, "usage: ringproc <iterations> <cuid> <receive_from_channel_desc> [<send_to_channel_desc> <final_channel_desc>]\n");
        fflush(stderr);
        return -1;
    }

    int iterations = atoi(argv[1]);
    dragonC_UID_t cuid = strtoul(argv[2], NULL, 0);

    dragonChannelSerial_t recv_chser;
    dragonChannelDescr_t recv_ch;
    dragonChannelRecvh_t recv_h;
    dragonChannelSerial_t send_chser;
    dragonChannelSerial_t final_chser;
    dragonChannelDescr_t send_ch;
    dragonChannelSendh_t send_h;
    dragonChannelDescr_t final_ch;
    dragonChannelSendh_t finalsend_h;
    dragonMemoryPoolDescr_t pool_descr;
    dragonMessage_t msg;
    char* send_ser_encoded;
    char* final_ser_encoded;
    size_t send_ser_len;

    /* This function is necessary for off-node communication and relies on the
    * Dragon run-time services to supply gateway channels in the
    * environment. Gateway channels are automatically supplied by Dragon
    * on multi-node allocations and this function works on both single
    * and multi-node allocations, though on single-node allocations it
    * does nothing. */

    dragonError_t err = dragon_channel_register_gateways_from_env();
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not register gateway channels from environment with err=%s\n", dragon_get_rc_string(err));
        fflush(stderr);
        return -1;
    }

    /*
    * When sending a message, the structure must be initialized first.
    */

    err = dragon_channel_message_init(&msg, NULL, NULL);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not init message with err=%s\n", dragon_get_rc_string(err));
        fflush(stderr);
        return -1;
    }

    /* A serialized channel descriptor is binary data which must be base64
    * encoded so it is valid ascii data before being passed around.
    * Dragon provides both base64 encoding and decoding for
    * interoperability between languages. */

    recv_chser.data = dragon_base64_decode(argv[3], strlen(argv[3]), &recv_chser.len);

    /* With a valid serialized descriptor you can attach to a channel. This
    * attach here occurs on an off-node channel (except in the one node
    * case). Whether off-node or on-node, attach works exactly the same.
    * */

    err = dragon_channel_attach(&recv_chser, &recv_ch);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not attach to receive channel with err=%s\n", dragon_get_rc_string(err));
        fprintf(stderr, "Converting '%s'\n", argv[3]);
        return -1;
    }

    /* The decode mallocs space. This frees any malloced code in the descriptor.
    * Be sure to only call this if there is malloced space stored in the
    * descriptor. */
```

```
    err = dragon_channel_serial_free(&recv_chser);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not free serialized channel descriptor with err=%s\n", dragon_get_rc_string(err));
        return -1;
    }

    /* The receive handle has optional attributes that are not supplied here. To
    * supply non-default attributes to the receive handle, call
    * dragon_channel_recv_attr_init first, then modify the attributes to
    * desired values and pass them as the third argument here. NULL means
    * to use the default attrs. */

    err = dragon_channel_recvh(&recv_ch, &recv_h, NULL);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not construct receive handle with err=%s\n", dragon_get_rc_string(err));
        fflush(stderr);
        return -1;
    }

    if (argc <= 4) {
        /* In most cases instance of this process, it creates a channel to send
        * the message to. To do this, the code must attach to a pool.
        * The default pool is already created, but users may also
        * create their own pools. The pool is an on-node resource
        * only, so it must exist where the channel is to be created.
        * There is a default pool on each node running under the
        * Dragon run-time services. */

        err = dragon_memory_pool_attach_from_env(&pool_descr, "DRAGON_DEFAULT_PD");
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not attach to memory pool with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

        /* We create our own send_to channel with the given cuid. Attributes
        * could be applied to the channel creation. NULL provides the
        * default attributes. To customize, call
        * dragon_channel_attr_init first, the customize and provide
        * them in place of NULL. */

        err = dragon_channel_create(&send_ch, cuid, &pool_descr, NULL);
        if (err != DRAGON_SUCCESS) {

            /* Notice the calls to dragon_get_rc_string which converts dragon
            * error codes into human readable strings. Also the
            * dragon_getlasterrstr provides useful traceback
            * information so you can see the origin of an error
            * should it occur. */

            fprintf(stderr, "Could not create send channel with err=%s\n", dragon_get_rc_string(err));
            fprintf(stderr, "Traceback: %s\n", dragon_getlasterrstr());
            fflush(stderr);
            return -1;
        }

        /*
        * Here we serialize the new channel and provide it on standard output.
        */

        err = dragon_channel_serialize(&send_ch, &send_chser);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not serialize send channel with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

        send_ser_encoded = dragon_base64_encode(send_chser.data, send_chser.len, &send_ser_len);

        err = dragon_memory_pool_detach(&pool_descr);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not detach to memory pool with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

        err = dragon_channel_serial_free(&send_chser);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not free serialized channel descriptor with err=%s\n", dragon_get_rc_string(err));
            return -1;
        }

    } else {
        /*
        * We were given a channel descriptor for the send channel and the final
        * send channel.
```

```
        */
        send_ser_encoded = argv[4];
        final_ser_encoded = argv[5];

        send_chser.data = dragon_base64_decode(send_ser_encoded, strlen(send_ser_encoded), &send_chser.len);

        err = dragon_channel_attach(&send_chser, &send_ch);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not attach to send channel with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

        err = dragon_channel_serial_free(&send_chser);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not free serialized channel descriptor with err=%s\n", dragon_get_rc_string(err));
            return -1;
        }

        final_chser.data = dragon_base64_decode(final_ser_encoded, strlen(final_ser_encoded), &final_chser.len);

        err = dragon_channel_attach(&final_chser, &final_ch);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not attach to final send channel with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

        /* The final channel is where to send the message when it has completed
        * its rounds on the ring. The final channel contents are read
        * by the Python bootstrap program to indicate that the test
        * has completed. */

        err = dragon_channel_serial_free(&final_chser);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not free final serialized channel descriptor with err=%s\n", dragon_get_rc_string(err));
            return -1;
        }

        err = dragon_channel_sendh(&final_ch, &finalsend_h, NULL);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not construct send handle for final channel with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

        err = dragon_chsend_open(&finalsend_h);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not open final send handle with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }
    }

    /*
    * This provides the newly created channel back to the caller of this code.
    */
    printf("%s\n", send_ser_encoded);
    fflush(stdout);

    /* The send handle is used to send message into a channel. Default attributes
    * are applied here. The send handle attributes can be customized by
    * calling dragon_channel_send_attr_init and providing in place of
    * NULL. */

    err = dragon_channel_sendh(&send_ch, &send_h, NULL);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not construct send handle with err=%s\n", dragon_get_rc_string(err));
        fflush(stderr);
        return -1;
    }

    /*
    * You must open send and receive handles before sending or receiving.
    */
    err = dragon_chsend_open(&send_h);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not open send handle with err=%s\n", dragon_get_rc_string(err));
        fflush(stderr);
        return -1;
    }

    err = dragon_chrecv_open(&recv_h);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not open receive handle with err=%s\n", dragon_get_rc_string(err));
        fflush(stderr);
        return -1;
```

```
    }

    int k;
    dragonChannelSendh_t* sendto_h = &send_h;

    for (k=0; k<iterations; k++) {
        /* Blocking receives may be given a timeout. This code blocks using the
        * default receive handle timeout which is to wait indefinitely. */

        err = dragon_chrecv_get_msg_blocking(&recv_h, &msg, NULL);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not receive message with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

        if ((argc > 4) && (k==iterations-1)) {
            /* On the last iteration for the origin process, write the message to
            * the final channel instead of back into the ring. */

            sendto_h = &finalsend_h;
        }

        /* Send the message on to its destination. Transfer of ownership means
        * that any pool allocation associated with the message will
        * be freed by the receiver. This works both on and off-node
        * since the transport agent will clean up the message in the
        * off-node case. */

        err = dragon_chsend_send_msg(sendto_h, &msg, DRAGON_CHANNEL_SEND_TRANSFER_OWNERSHIP, NULL);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not send message with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }
    }

    /*
    * Send and receive handles should be closed when no longer needed.
    */

    err = dragon_chsend_close(&send_h);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not close send handle with err=%s\n", dragon_get_rc_string(err));
        fflush(stderr);
        return -1;
    }

    err = dragon_chrecv_close(&recv_h);
    if (err != DRAGON_SUCCESS) {
        fprintf(stderr, "Could not close receive handle with err=%s\n", dragon_get_rc_string(err));
        fflush(stderr);
        return -1;
    }
    if (argc <= 4) {

        /* Channels should be destroyed when no longer needed. Since the program
        * is ending, technically this would be cleaned up
        * automatically once the Dragon run-time services exit, but
        * better to be explicit about it in this example. */

        err = dragon_channel_destroy(&send_ch);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not destroy send channel with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

        /* To be complete, we'll detach from the pool. But again, this is done
        * automatically during cleanup when Dragon run-time services
        * exit. */

        err = dragon_memory_pool_detach(&pool_descr);
        if (err != DRAGON_SUCCESS) {
            fprintf(stderr, "Could not detach from the default pool with err=%s\n", dragon_get_rc_string(err));
            fflush(stderr);
            return -1;
        }

    }

    return 0;
}
```