C. CLASSES AND SUPPORTING TYPES

C.1 Classes

C.1.1 AttributeHandleSet

RTI 1.3-NG

ABSTRACT

This class implements an unordered collection of unique attribute handles. Instances of this class are typically accompanied by an object-class handle or an object-instance handle to provide a context for the attribute handles.

SYNOPSIS

```
#include <RTI.hh>

class RTI::AttributeHandleSet {
public:
   virtual
      ~AttributeHandleSet( );

   virtual RTI::ULong
      size( ) const;

  virtual RTI::AttributeHandle
      getHandle(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds
         );

   virtual void
      add(RTI::AttributeHandle h)
         throw (
            RTI::ArrayIndexOutOfBounds
            RTI::AttributeNotDefined
         );

   virtual void
      remove(RTI::AttributeHandle h)
         throw (
            RTI::AttributeNotDefined
         )
  virtual void
      empty();

  virtual RTI::Boolean
      isEmpty() const;

  virtual RTI::Boolean
      isMember(RTI::AttributeHandle h) const;
};
```

DESCRIPTION

This class implements an unordered collection of unique attribute handles, i.e. a *set*. Such a set may be used in conjunction with an object-class handle to denote a set of class-attributes (e.g., when using declaration management services such as `publishObjectClass()`). A set may also be used in conjunction with an object-instance identifier to denote a set of instance-attributes (e.g., when using object management services such as `requestObjectAttributeValueUpdate()`). When used in the later context, the handles in the set are assumed to refer to attributes of the class by which the object is known to the federate.

An attribute-handle set exists independently of any object-class or object-instance context. Any value in the range of attribute handles (i.e., zero through *MAX_ATTRIBUTES_PER_CLASS* – 1) may be included in a set. However, when a set is used as an argument to an RTI service method, it is expected that the set will only contain handles that are valid in the object-class or object-instance context related to the service invocation.

Instances of this class may only be constructed using the `AttributeHandleSetFactory` class. Instances should be destroyed using the `delete` operator when no longer needed.

METHODS

~AttributeHandleSet( )

The class destructor is implicitly invoked whenever an instance is destroyed using the `delete` operator. The destructor deallocates the memory associated with the class instance.

add( )

This method adds the specified attribute handle to the set. If the handle is already present the method will continue to add addition instances of the same handle to the set. Neither the `ArrayIndexOutOfBounds` nor the AttributeNotDefined exceptions are not thrown in the RTI1.3-NG implementation.

Depending on the object-class context in which an attribute handle set is used, not all attribute handles in this range may be valid. Handles that are invalid for a particular object class may be added to the set but will result in an exception when the set is passed as an argument to an RTI service.

empty( )

This method removes all attribute handles from the set.

getHandle( )

This method treats the handle set as if it were an array containing all the handles in the set sorted in ascending order. The method returns the handle corresponding to offset *i* in the array. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

The intent of this method is to facilitate loops of the following sort:

```
RTI::ULong count = ahset.size();
for (RTI::ULong index = 0; index <
       count; ++index) {
            RTI::AttributeHandle h =
               ahset.getHandle(index);
            // some action based on h
}
```

The `isMember()` method may be used to test for the presence of a particular handle without iterating through the entire set.

isEmpty( )

This method returns *RTI::RTI_FALSE* if the set contains at least one attribute handle, otherwise *RTI::RTI_TRUE*.

isMember( )

This method may be used to quickly test for the presence of a particular handle in the set. It returns *RTI::RTI_TRUE* if the set contains the specified handle, otherwise *RTI::RTI_FALSE*. The caller should ensure that the argument is within the range of valid attribute handles.

The `getHandle()` method may be used to construct loops iterating over all the handles in a set.

remove( )

This method removes the specified attribute handle from the set. If the handle is not present, this results in a no-op. The AttributeNotDefined exception is not thrown in the RTI1.3-NG implementation.

size( )

This method returns the count of attribute handles currently in the set.

RTI 1.3-NG NOTES

RTI 1.3-NG makes `AttributeHandleSet` an abstract class,

meaning that instances may only be constructed using the `AttributeHandleSetFactory`. Assigning values to instances using the equal-sign operator or copy constructor will no longer work. The `setUnion()`, `setIntersection()`, and `removeSetIntersection()` methods are no longer available, as they relied on the copy constructor to return new instances. The `encodedLength()`, `encode()`, and `decode()` methods and the overloaded stream operators, intended for RTI internal use, are not publicly available.

SEE ALSO

RTI::AttributeHandleValuePairSet
RTI::AttributeHandleSetFactory
RTI::RTIambassador::
getAttributeHandle()
getAttributeName()
getObjectClassHandle()

C.1.2 AttributeHandleValuePairSet

RTI 1.3-NG

ABSTRACT

This class implements an ordered collection of pairs containing an attribute handle and an untyped value. Instances of this class are typically accompanied by an object-instance handle to provide a context for the attribute handles.

SYNOPSIS

```
#include <RTI.hh>

class RTI::AttributeHandleValuePairSet {
public:
   virtual
      ~AttributeHandleValuePairSet( );

   virtual RTI::ULong
      size() const;

   virtual RTI::Handle
      getHandle(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds
         );

   virtual RTI::ULong
      getValueLength(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds
         );

   virtual void
      getValue(
         RTI::ULong  i,
         char*       buff,
         RTI::ULong& valueLength
      ) const
      throw (
            RTI::ArrayIndexOutOfBounds
      );

   virtual char *
      getValuePointer(
         RTI::ULong  i,
         RTI::ULong& valueLength
      ) const
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual RTI::TransportType
      getTransportType(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds,
            RTI::InvalidHandleValuePairSetContext
         );

   virtual RTI::OrderType
      getOrderType(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds,
            RTI::InvalidHandleValuePairSetContext
         );

   virtual RTI::Region*
      getRegion(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds,
            RTI::InvalidHandleValuePairSetContext
         );

   virtual void
      add(
         RTI::Handle      h,
         const char*      buff,
         RTI::ULong       valueLength
      )
      throw (
         ValueLengthExceeded,
         ValueCountExceeded
      );

   virtual void
      remove(RTI::Handle h)
         throw (
            ArrayIndexOutOfBounds
         );

   virtual void
      moveFrom(
         const AttributeHandleValuePairSet& ahvps,
         RTI::ULong&               i
      )
      throw (
         ValueCountExceeded,
         ArrayIndexOutOfBounds
      );

   virtual void
      empty ( ) ;

   virtual RTI::ULong
      start() const;

   virtual RTI::ULong
      valid(RTI::ULong i) const;

   virtual RTI::ULong
      next(RTI::ULong i) const;
};
```

DESCRIPTION

This class implements an ordered collection of handle-value pairs. Instances of this class are used to convey federation data associated with an object update. When used in such a fashion, the attribute handles in the collection are assumed to refer to attributes of the object class by which the object is known to the federate.

A handle-value pair collection exists independently of any object-instance context. Any value in the range of attribute handles (i.e., zero through *MAX_ATTRIBUTES_PER_CLASS* – 1) may be associated with values in a collection. However, when a collection is used as an argument to an RTI service method, it is expected that the collection will only contain associations for handles that are valid in the object-instance context related to the service invocation.

The RTI treats values associated with handles as opaque sequences of bytes: the values are relayed from sender to receiver with no conversion or interpretation. It is the responsibility of the federation developer to ensure that federates have a common understanding of the values being conveyed. In a non-homogenous environment, endian-conversion and other data-representation adjustments may be necessary.

Instances of this class may only be constructed using the `AttributeSetFactory` class. Instances should be destroyed using the `delete` operator when no longer needed.

The ordering associated with pairs in the collection is unimportant unless the same handle is associated with multiple values. In this case, the `remove()` method only removes the first pair encountered that is associated with the specified attribute handle, based on a sequential iteration through the collection. The federate developer is discouraged from relying on the ordering associated with pairs in a handle-value pair collection.

METHODS

~AttributeHandleValuePairSet( )

The class destructor is implicitly invoked whenever an instance is destroyed using the `delete` operator. The destructor deallocates the memory associated with the class

instance.

add( )

This method inserts a new handle-value pair consisting of the specified handle and caller-supplied buffer. A copy of the buffer is stored in the collection; the caller is free to overwrite the contents of the pointer at any time.

If the insertion would result in the number of pairs in the collection exceeding the limit specified at its creation, the `ValueCountExceeded()` exception is thrown. The `ValueLengthExceeded()` exception may be thrown if not enough resources are available to copy the caller-supplied buffer. The RTI does not impose an arbitrary limit on the size of values; they are limited only by system resources.

If this method is repeatedly invoked with the same attribute handle, multiple pairs will be inserted, each associated with the same attribute handle. Federate developers are discouraged from exploiting this behavior.

empty( )

This method removes all handle-value pairs from the collection.

getHandle( )

This method treats the handle-value pair collection as an array containing all pairs in the set in the order in which they were inserted. The method returns the handle corresponding to offset *i* in the array. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

The intent of this method is to facilitate loops of the following sort:

```
RTI::ULong count = ahvpset.size();
for (RTI::ULong index = 0; index <
       count; ++index) {
            RTI::AttributeHandle h =
               ahvpset.getHandle(index);
            // some action based on h
}
```

getOrderType( )

This method may be used by the receiver of a handle-value pair set to obtain a handle to the ordering service category that was used to *send* the set. (If the federate is not time-constrained, all events will be considered receive-ordered regardless of the ordering type associated with the set.)

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. RTI 1.3 does not deliver attributes with different ordering policies in the same set, so this method will return the same handle for any valid index in the set. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

This method is only valid for sets that represent incoming events. An attempt to invoke this method on a set that was created by the local federate will result in an `InvalidHandleValuePairSetContext` exception.

getRegion( )

This method may be used by the receiver of a handle-value pair set to obtain a handle to the ordering service category that was used to send the set.

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. The RTI does not deliver attributes with different regions in the same set, so this method will return the same region for any valid index in the set. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

This method is only valid for sets that represent incoming events. An attempt to invoke this method on a set that was created by the local federate will result in an `InvalidHandleValuePairSetContext` exception.

getTransportType( )

This method may be used by the receiver of a handle-value pair set to obtain a handle to the transportation service category that was used to send the set.

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. The RTI does not deliver attributes with different transportation policies in the same set, so this method will return the same handle for any valid index in the set. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

This method is only valid for sets that represent incoming events. An attempt to invoke this method on a set that was created by the local federate will result in an `InvalidHandleValuePairSetContext` exception.

getValue( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. The method copies the value corresponding to offset *i* in the array into the caller-supplied buffer, *buff*. The length in bytes of the copied value is returned via the *valueLength* out-parameter. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

The `getValue()` method assumes that the caller-supplied buffer is large enough to hold the value. The `getValueLength()` method should be consulted prior to invoking `getValue()` to ensure that the supplied buffer is sufficiently large.

This method is intended to be used in conjunction with `getHandle()` to iterate through a set.

getValueLength( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. The method returns the handle corresponding to offset *i* in the array. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

This method should be called prior to invoking `getValue()` in order to determine the size of the buffer that is required.

getValuePointer( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. This method returns a pointer to the contents of the value corresponding to offset *i* in the array. The length in bytes of the returned value is returned via the *valueLength* out-parameter. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is

thrown.

This method is intended to be used in conjunction with `getHandle()` to iterate through a set.

If the caller intends to use the pointer returned by this method to modify the contents of a value previously inserted into the set, the new contents copied into the address must have the same length as the original contents.

For instances that have been passed to the federate as an argument to the `reflectAttributeValues()` service, the pointer returned by this method is only valid for the duration of the callback. The federate must make a copy of this memory if it requires the value to persist beyond the return from the service invocation.

moveFrom( )

This method transfers a handle-value pair from one set to another without performing any memory copying. Given handle-value pair sets *to* and *from* and an index *i*, this method is a more efficient shorthand for the following:

```
RTI::ULong size;
char* value;
RTI::AttributeHandle handle;
value = from.getValuePointer(i, size);
handle = from.getHandle();
to.add(handle, value, size);
from.remove(handle); // assuming only one
                        instance of handle
                        in the set
```

Note that this method modifies its first argument, although it is declared *const* in the signature.

next( )

This method returns the value of its argument plus one.

The `start()`, `valid()`, and `next()` methods may be used to iterate over all the pairs in a set as follows:

```
RTI::ULong iter;
for (iter = ahvpset.start();
     ahvpset.valid(iter);
     ahvpset.next(iter++))
        // some action based on index iter
```

remove( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. This method removes only the first pair associated with attribute handle *h* from the array, based on a zero-based sequential iteration. If *h* does not occur in the set, the `ArrayIndexOutOfBounds` exception is thrown.

The last pair in the array (i.e., the most recently inserted item) is moved into the hole vacated by the removed item.

size( )

This method returns the count of attribute handle-value pairs currently in the set. Multiple values associated with the same handle each count towards the total size.

start( )

This method returns zero if the set is non-empty and an invalid attribute handle otherwise.

The `start()`, `valid()`, and `next()` methods may be used to iterate over all the pairs in a set as follows:

```
RTI::ULong iter;
for (iter = ahvpset.start();
     ahvpset.valid(iter);
     ahvpset.next(iter++))
        // some action based on index iter
```

valid( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. The method returns a non-zero value if its argument *i* is a valid index in the array (i.e., $i < 0$ or $i \geq size$), otherwise the method returns zero.

The `start()`, `valid()`, and `next()` methods may be used to iterate over all the pairs in a set as follows:

```
RTI::ULong iter;
for (iter = ahvpset.start();
     ahvpset.valid(iter);
     ahvpset.next(iter++))
        // some action based on index iter
```

RTI 1.0 MIGRATION NOTES

In RTI 1.0, attribute handle- and parameter handle-value pair sets were interchangeable (they were `typedef`'d to the same class.) In RTI 1.3 this is no longer the case.

The `getValuePointer()` allows zero-copy access to values in a handle-value pair collection. Where possible, use this method instead of `getValue()` for more efficient data access.

RTI 1.0 limited values to a maximum size of 4096, as `define`'d by the `MAX_BYTES_PER_VALUE` constant. There is no arbitrary limit on the size of values in RTI 1.3. Consequently, it is necessary to call the `getValueLength()` method before using `getValue()` to ensure that the buffer supplied to `getValue()` is sufficiently large. (Alternatively, use the `getValuePointer()` method instead.)

The `getTransportType()`, `getOrderType()`, and `getRegion()` methods are new to RTI 1.3. They may be used by the receiver of a reflection to obtain information about the event.

SEE ALSO

RTI::AttributeHandleSet
RTI::AttributeSetFactory
RTI::ParameterHandleValuePairSet
RTI::Region
RTI::RTIambassador
getAttributeHandle()
getAttributeName()
getObjectClassHandle()
getOrderingName()
getTransportationName()

RTI 1.3-NG

ABSTRACT

This class is the superclass of all the exceptions used by the RTI. It contains data members used to communicate details of the exception.

SYNOPSIS

```
class RTI::Exception {
public:
   RTI::ULong _serial;

   char *_reason;

   const char *_name;

   Exception (const char *reason);
```

```
        Exception (
            RTI::ULong serial,
                const char *reason=NULL
        );

    Exception (const RTI::Exception &toCopy);

    virtual ~Exception( );

    Exception & operator = (const Exception &);

    friend ostream& operator<< (
        ostream &,
        RTI::Exception *
    );
};
```

DESCRIPTION

The `Exception` class is a simple class containing a few data members and some convenience constructors and operators. It is subclassed by the various RTI exception classes to allow different exception handlers to be written for different types of errors.

DATA MEMBERS

_name

C.1.3 Exception

the class name of the specific subclass of exception thrown

_reason

a textual description of the reason the exception was thrown (or NULL)

_serial

a serial number uniquely identifying the place in the RTI code where the exception originated; include this number when reporting problems to technical support

METHODS

~Exception( )

The class destructor is implicitly invoked whenever an instance of the class is deallocated. The destructor deallocates the memory associated with the instance.

Exception (const char *)

This constructor creates a new instance based on a string reason (or NULL.) A new copy is made of the string argument, if provided.

Exception (const RTI::Exception &)

The copy constructor creates a new instance that is identical to an existing instance. A copy of the *_reason* string (if present) is made for the new instance.

OPERATORS

operator =

The assignment operator reinitializes an instance based on the value of another instance. A copy of the *_reason* string (if present) is made for the instance being assigned to.

operator<<

The stream operator writes the class name, serial number, and reason associated with the instance to the stream.

C.1.4 FederateHandleSet

RTI 1.3-NG

ABSTRACT

This class represents an unordered collection of federate handles.

SYNOPSIS

```
#include <RTI.hh>

class RTI::FederateHandleSet {
public:
   virtual RTI::ULong
      size( ) const;

   virtual RTI::FederateHandle
      getHandle(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds
         );

   virtual void
      add(RTI::FederateHandle h)
         throw (
            RTI::ValueCountExceeded
         );

   virtual void
      remove(RTI::FederateHandle h)
         throw (
            RTI::ArrayIndexOutOfBounds
         );

   virtual void
      empty( );

   virtual RTI::Boolean
      isMember(RTI::FederateHandle h) const;
};
```

DESCRIPTION

This class implements an unordered collection of federate handles. Instances of this class are used to denote subsets an active federation. Such collections exist independently of the federate handles actually in use in a federation at a particular instant. However, when used as arguments to RTI services, they are expected to only contain handles that are valid in the context of the current active federation.

Federate handles of remote federates may be obtained by subscribing to the *Manager.Federate* object class maintained by the Management Object Model.

METHODS

add

This method adds the specified handle to the collection. The value added need not correspond to a currently active federate in the federation. Invoking this method multiple times with the same argument will result in the same federate handle appearing multiple times in the collection.

If adding a handle would result in the maximum collection size specified during the creation of the instance being exceeded, the `ValueCountExceeded` exception is thrown.

empty

This method removes all handles currently in the collection.

getHandle

This method treats the handle set as if it were an array containing all the handles in the set appearing in arbitrary order. The method returns the handle corresponding to offset *i* in the array. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

The intent of this method is to facilitate loops of the following sort:

```
RTI::ULong count = fhset.size();
for (RTI::ULong index = 0; index <
       count; ++index) {
            RTI::FederateHandle h =
               fhset.getHandle(index);
            // some action based on h
}
```

The `isMember()` method may be used to test for the presence of a particular handle without iterating through the entire set.

isMember

This method returns *RTI::RTI_TRUE* if any instances of the specified handle are present in the collection, otherwise it returns *RTI::RTI_FALSE*.

The `getHandle()` method may be used to iterate over all handles in the collection.

remove

This method removes a single instance of the specified handle from the collection. If the specified handle is not present at all, the `ArrayIndexOutOfBounds` exception is thrown.

This method should not be invoked while an iteration based on `getHandle()` is in progress.

size

This method returns the current count of handles in the set. Multiple instances of the same handle count multiple times towards this count.

SEE ALSO

RTI::FederateHandleSet

RTI 1.3-NG

C.1.5 FedTime

ABSTRACT

Instances of this class represent points on the federation logical time axis.

SYNOPSIS

```
#include <RTI.hh>

class RTI::FedTime {
public:
   virtual
      ~FedTime( );

   virtual void
      setZero( );

   virtual RTI::Boolean
      isZero( )

   virtual void
      setEpsilon( )

   virtual void
      setPositiveInfinity( );

   virtual RTI::Boolean
      isPositiveInfinity( );

   virtual RTI::FedTime&
      operator+= (const RTI::FedTime&)
         throw (
            InvalidFederationTime
         );

   virtual RTI::FedTime&
      operator-= (const RTI::FedTime&)
         throw (
            InvalidFederationTime
         );

   virtual RTI::Boolean
      operator<= (const RTI::FedTime&) const
         throw (
            InvalidFederationTime
         );

   virtual RTI::Boolean
      operator< (const RTI::FedTime&) const
         throw (
            InvalidFederationTime
         );

   virtual RTI::Boolean
      operator>= (const RTI::FedTime&) const
         throw (
            InvalidFederationTime
         );

   virtual RTI::Boolean
      operator> (const RTI::FedTime&) const
         throw (
            InvalidFederationTime
         );

   virtual RTI::Boolean
      operator== (const RTI::FedTime&) const
         throw (
            InvalidFederationTime
         );

   virtual RTI::FedTime&
      operator= (const RTI::FedTime&)
         throw (
            InvalidFederationTime
         );

   virtual int
      encodedLength( ) const;

   virtual void
      encode(char *buff) const;

   virtual int
      getPrintableLength( ) const;

   virtual void
      getPrintableString(char*);
};
```

DESCRIPTION

Instances of this class represent points on the federation logical time axis. The methods of this class provide a means for setting an instance equal to certain “special” points in the federation time axis. The operators provide a means for adjusting or comparing the axis point represented by an instance with that of a second instance.

`FedTime` instances representing arbitrary points on the federation logical time axis may be constructed using the `FedTimeFactory::decode()` method. The argument to this method should be a character pointer (char *).

Conversely, the precise value associated with a `FedTime` instance may be queried using the `encode()` method. The argument to this method should be a character pointer (char *) into which the value associated with the instance will be marshaled.

METHODS

~FedTime

The destructor is implicitly invoked whenever an instance of the class is deallocated. The destructor frees any resources associated with the class instance.

encode

This method marshals the `FedTime` instance into a buffer suitable for transmission over the network and subsequent decoding using the `FedTimeFactory` class. The marshaled instance will occupy the number of bytes returned by the `encodedLength()` method.

encodedLength

This method returns the length in bytes occupied by the instance when marshaled using `encode()`.

getPrintableLength

This method is not implemented; it always returns zero.

getPrintableString

This method is not implemented; it performs a no-op.

isPositiveInfinity

This method returns *RTI::RTI_TRUE* if the logical time represented by the instance is positive infinity, otherwise it returns *RTI::RTI_FALSE*. Positive infinity represents the upper bound of the federation logical time axis. The federation lower-bound time-stamp for a federation with no time-regulating federates is equal to positive infinity.

isZero

This method returns *RTI::RTI_TRUE* if the logical time represented by the instance is zero, otherwise it returns *RTI::RTI_FALSE*. Zero represents the lower bound of the federation logical time axis.

setEpsilon

This method sets the logical time represented by the instance equal to epsilon, or 10^{-9}. Epsilon is the default value for

federate lookahead.

setPositiveInfinity

This method sets the logical time represented by the instance equal to positive infinity. Positive infinity represents the upper bound of the federation logical time axis.

setZero

This method sets the logical time represented by the instance equal to zero. Zero represents the lower bound of the federation logical time axis.

OPERATORS

operator+=

This operator increments the logical time represented by the instance based on the value of a second instance.

operator<

This operator evaluates to true if and only if the logical time represented by the left-hand operand strictly precedes the logical time represented by the right-hand operand.

operator<=

This operator evaluates to true if and only if the logical time represented by the left-hand operand precedes or equals the logical time represented by the right-hand operand..

operator=

This operator sets the logical time represented by the instance based on the value of a second instance.

operator-=

This operator decrements the logical time represented by the instance based on the value of a second instance.

operator==

This operator evaluates to true if and only if the logical time represented by the left-hand operand equals the logical time represented by the right-hand operand..

operator>

This operator evaluates to true if and only if the logical time represented by the left-hand operand strictly follows the logical time represented by the right-hand operand.

operator>=

This operator evaluates to true if and only if the logical time represented by the left-hand operand follows or equals the logical time represented by the right-hand operand.

SEE ALSO

RTI::FedTimeFactory

C.1.6 ParameterHandleValuePairSet

RTI 1.3-NG

ABSTRACT

This class implements an ordered collection of pairs containing a parameter handle and an untyped value. Instances of this class are typically accompanied by an interaction-class handle to provide a context for the parameter handles.

SYNOPSIS

```
#include <RTI.hh>

class ParameterHandleValuePairSet {
public:
   virtual
      ~ParameterHandleValuePairSet( )

   virtual RTI::ULong
      size( ) const;

   virtual RTI::Handle
      getHandle(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds
         );

   virtual RTI::ULong
      getValueLength(RTI::ULong i) const
         throw (
            RTI::ArrayIndexOutOfBounds
         );

   virtual void
      getValue(
         RTI::ULong i,
         char*      buff,
         RTI::ULong&     valueLength
      ) const
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual char *
      getValuePointer(
         RTI::ULong i,
         RTI::ULong&     valueLength
      ) const
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual RTI::TransportType
      getTransportType( ) const
      throw (
         RTI::InvalidHandleValuePairSetContext
      );

   virtual RTI::OrderType
      getOrderType( ) const
      throw (
         RTI::InvalidHandleValuePairSetContext
      );

   virtual RTI::Region *
      getRegion( ) const
      throw (
         RTI::InvalidHandleValuePairSetContext
      );

   virtual void
      add(
         RTI::Handle      h,
         const char*      buff,
         RTI::ULong       valueLength
      )
      throw (
         RTI::ValueLengthExceeded,
         RTI::ValueCountExceeded
      );

   virtual void
      remove(RTI::Handle h)
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual void
      moveFrom(
         const RTI::ParameterHandleValuePairSet&
            phvps,
         RTI::ULong& i
      )
      throw (
         RTI::ValueCountExceeded,
         RTI::ArrayIndexOutOfBounds
      );

   virtual void
      empty( );

   virtual RTI::ULong
      start( ) const;

   virtual RTI::ULong
      valid(RTI::ULong i) const;

   virtual RTI::ULong
      next(RTI::ULong i) const;
};
```

SYNOPSIS

This class implements an ordered collection of handle-value pairs. Instances of this class are used to convey federation data associated with an interaction. When used in such a fashion, the parameter handles in the collection are assumed to refer to parameters of the class of the interaction being sent or reflected.

A handle-value pair collection exists independently of any interaction-class context. Any value in the range of parameter handles (i.e., zero through *MAX_PARAMETERS_PER_CLASS* – 1) may be associated with values in a collection. However, when a collection is used as an argument to an RTI service method, it is expected that the collection will only contain associations for handles that are valid in the interaction-class context related to the service invocation.

The RTI treats values associated with handles as opaque sequences of bytes: the values are relayed from sender to receiver with no conversion or interpretation. It is the responsibility of the federation developer to ensure that federates have a common understanding of the values being conveyed. In a homogenous environment, endian-conversion and other data-representation adjustments may be necessary.

Instances of this class may only be constructed using the `ParameterSetFactory` class. Instances should be destroyed using the `delete` operator when no longer needed.

The ordering associated with pairs in the collection is unimportant unless the same handle is associated with multiple values. In this case, the `remove()` method only removes the first pair encountered that is associated with the specified parameter handle, based on a sequential iteration through the collection. The federate developer is discouraged from relying on the ordering associated with pairs in a handle-value pair collection.

METHODS

~ParameterHandleValuePairSet( )

The class destructor is implicitly invoked whenever an instance is destroyed using the `delete` operator. The destructor deallocates the memory associated with the class instance.

add( )

This method inserts a new handle-value pair consisting of the

specified handle and caller-supplied buffer. A copy of the buffer is stored in the collection; the caller is free to overwrite the contents of the pointer at any time.

If the insertion would result in the number of pairs in the collection exceeding the limit specified at its creation, the `ValueCountExceeded()` exception is thrown. The `ValueLengthExceeded()` exception may be thrown if not enough resources are available to copy the caller-supplied buffer. The RTI does not impose an arbitrary limit on the size of values; they are limited only by system resources.

If this method is repeatedly invoked with the same parameter handle, multiple pairs will be inserted, each associated with the same parameter handle. Federate developers are discouraged from exploiting this behavior.

empty( )

This method removes all handle-value pairs from the collection.

getHandle( )

This method treats the handle-value pair collection as an array containing all pairs in the set in the order in which they were inserted. The method returns the handle corresponding to offset *i* in the array. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

The intent of this method is to facilitate loops of the following sort:

```
RTI::ULong count = phvpset.size();
for (RTI::ULong index = 0; index <
       count; ++index) {
            RTI::ParameterHandle h =
               phvpset.getHandle(index);
            // some action based on h
}
```

getOrderType( )

This method may be used by the receiver of a handle-value pair set to obtain a handle to the ordering service category that was used to *send* the set. (If the federate is not time-constrained, all events will be considered receive-ordered regardless of the ordering type associated with the set.)

This method is only valid for sets that represent incoming events. An attempt to invoke this method on a set that was created by the local federate will result in an `InvalidHandleValuePairSetContext` exception.

getRegion( )

This method may be used by the receiver of a handle-value pair set to obtain a handle to the ordering service category that was used to send the set.

This method is only valid for sets that represent incoming events. An attempt to invoke this method on a set that was created by the local federate will result in an `InvalidHandleValuePairSetContext` exception.

getTransportType( )

This method may be used by the receiver of a handle-value pair set to obtain a handle to the transportation service category that was used to send the set.

This method is only valid for sets that represent incoming events. An attempt to invoke this method on a set that was created by the local federate will result in an `InvalidHandleValuePairSetContext` exception.

getValue( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. The method copies the value corresponding to offset *i* in the array into the caller-supplied buffer, *buff*. The length in bytes of the copied value is returned via the *valueLength* out-parameter. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

The `getValue()` method assumes that the caller-supplied buffer is large enough to hold the value. The `getValueLength()` method should be consulted prior to invoking `getValue()` to ensure that the supplied buffer is sufficiently large.

This method is intended to be used in conjunction with `getHandle()` to iterate through a set.

getValueLength( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. The method returns the handle corresponding to offset *i* in the array. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

This method should be called prior to invoking `getValue()` in order to determine the size of the buffer that is required.

getValuePointer( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. This method returns a pointer to the contents of the value corresponding to offset *i* in the array. The length in bytes of the returned value is returned via the *valueLength* out-parameter. If *i* is not a valid index into the array (i.e., $i < 0$ or $i \geq size$), the `ArrayIndexOutOfBounds` exception is thrown.

This method is intended to be used in conjunction with `getHandle()` to iterate through a set.

If the caller intends to use the pointer returned by this method to modify the contents of a value previously inserted into the set, the new contents copied into the address must have the same length as the original contents.

For instances that have been passed to the federate as an argument to the `receiveInteraction()` service, the pointer returned by this method is only valid for the duration of the callback. The federate must make a copy of this memory if it requires the value to persist beyond the return from the service invocation.

moveFrom( )

This method transfers a handle-value pair from one set to another without performing any memory copying. Given handle-value pair sets *to* and *from* and an index *i*, this method is a more efficient shorthand for the following:

```
RTI::ULong size;
char* value;
RTI::ParameterHandle handle;
value = from.getValuePointer(i, size);
handle = from.getHandle();
to.add(handle, value, size);
```

```
from.remove(handle); // assuming only one
                        instance of handle
                        in the set
```

Note that this method modifies its first argument, although it is declared *const* in the signature.

next( )

This method returns the value of its argument plus one.

The `start()`, `valid()`, and `next()` methods may be used to iterate over all the pairs in a set as follows:

```
RTI::ULong iter;
for (iter = phvpset.start();
     phvpset.valid(iter);
     phvpset.next(iter))
        // some action based on index iter
```

remove( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. This method removes only the first pair associated with parameter handle *h* from the array, based on a zero-based sequential iteration. If *h* does not occur in the set, the `ArrayIndexOutOfBounds` exception is thrown.

The last pair in the array (i.e., the most recently inserted item) is moved into the hole vacated by the removed item.

size( )

This method returns the count of parameter handle-value pairs currently in the set. Multiple values associated with the same handle each count towards the total size.

start( )

This method returns zero if the set is non-empty and an invalid parameter handle otherwise.

The `start()`, `valid()`, and `next()` methods may be used to iterate over all the pairs in a set as follows:

```
RTI::ULong iter;
for (iter = phvpset.start();
     phvpset.valid(iter);
     phvpset.next(iter))
        // some action based on index iter
```

valid( )

This method treats the handle-value pair set as an array containing all pairs in the set in the order in which they were inserted. The method returns a non-zero value if its argument *i* is a valid index in the array (i.e., $i < 0$ or $i \geq size$), otherwise the method returns zero.

The `start()`, `valid()`, and `next()` methods may be used to iterate over all the pairs in a set as follows:

```
RTI::ULong iter;
for (iter = phvpset.start();
     phvpset.valid(iter);
     phvpset.next(iter))
        // some action based on index iter
```

SEE ALSO

RTI::ParameterSetFactory
RTI::AttributeHandleValuePairSet
RTI::Region
RTI::RTIambassador
getInteractionClassHandle()
getOrderingName()
getParameterHandle()
getParameterName()
getTransportationName()

C.1.7 Region

RTI 1.3-NG

ABSTRACT

An instance of this class represents a subspace of a federation routing space. The subspace is defined by the union of one or more *extents*, each of which is characterized by a lower and upper bound in every dimension.

SYNOPSIS

```
#include <RTI.hh>

class RTI::Region {
public:

   virtual
      ~Region( );

   virtual RTI::ULong
      getRangeLowerBound(
         RTI::ExtentIndex     theExtent,
         RTI::DimensionHandle theDimension
      ) const
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual RTI::ULong
      getRangeUpperBound(
         RTI::ExtentIndex     theExtent,
         RTI::DimensionHandle theDimension
      ) const
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual void
      setRangeLowerBound(
         RTI::ExtentIndex     theExtent,
         RTI::DimensionHandle theDimension,
         RTI::ULong           theLowerBound
      )
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual void
      setRangeUpperBound(
         RTI::ExtentIndex     theExtent,
         RTI::DimensionHandle theDimension,
         RTI::ULong           theUpperBound
      )
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual RTI::SpaceHandle
      getSpaceHandle( ) const
      throw ( );

   virtual RTI::ULong
      getNumberOfExtents( ) const
      throw ( );

   virtual RTI::ULong
      getRangeLowerBoundNotificationLimit(
         RTI::ExtentIndex     theExtent,
         RTI::DimensionHandle theDimension
      ) const
      throw (
         RTI::ArrayIndexOutOfBounds
      );

   virtual RTI::ULong
      getRangeUpperBoundNotificationLimit(
         RTI::ExtentIndex     theExtent,
         RTI::DimensionHandle theDimension
      ) const
      throw (
         RTI::ArrayIndexOutOfBounds
      );
};
```

DESCRIPTION

A region instance represents a subspace of the routing space with which it was associated upon instantiation. This subspace is defined by the union of all extents comprising the region. Each extent is characterized by a lower bound and an upper bound in each dimension of the routing space. For an *n*-dimensional routing space, a single extent consists of a vector of pairs of real numbers

$$(L_0, U_0), (L_1, U_1), \ldots, (L_{n-1}, U_{n-1})$$

such that for all *i* in the range [0,n-1],

$$\texttt{MIN\_EXTENT} \le L_i \le U_i < \texttt{MAX\_EXTENT}$$

A given point in the routing space,

$$(P_0, P_1, \ldots, P_{n-1})$$

is contained in the extent if and only if for all *i* in the range [0,n-1],

$$L_i \le P_i \le U_i$$

L_i is called the *lower bound* of the extent in dimension *i*. U_i is called the *upper bound* of the extent in dimension *i*.

A point is contained in a region if and only if there exists an extent of the region such that the point is contained in the extent.

Two regions of the same routing space are said to intersect if and only if there exists a point that is contained in both regions.

Each region has an implicit *notification subspace* that is derived from the region's extents and from the routing space configuration specified in the RID file. When a region is used for subscription, filtering of events delivered to the federate is based on the region's notification subspace. As such, the notification subspace includes the entire subspace defined by the extents of the region.

The notification subspace of a region is the union of the notification subspaces of its constituent extents. When the region is used for subscription, an event of the subscribed type is considered relevant to the federate if the notification subspace intersects the region associated with the event by the sender.

Changes made to a `Region` instance will not take effect until the instance is the subject of a `notifyAboutRegionModification()` service invocation. A newly created `Region` instance spans the entire routing space in all extents (i.e., all upper bounds are equal to `MAX_EXTENT` and all lower bounds are equal to `MIN_EXTENT`). The federate should make the desired modifications to the region's extents and recommit the region instance using `notifyAboutRegionModification()`. If any subsequent changes are made to the instance, this service should again be used to inform the RTI of the new extents.

The methods of the `Region` class do not enforce correctness of the region instance. That is, ranges may be specified such that the bounds are not valid extent values or such that the lower bound is greater than the upper bound. Such semantic errors are permitted by the `Region` object itself but will result in an exception when an incorrect instance is used as an argument to an RTI service method.

EXAMPLE

As an example, consider a routing space consisting of two dimensions, *x* and *y*. The complete routing space can be thought of as a box in two dimensions characterized by the points

- `(MIN_EXTENT, MIN_EXTENT)` [**lower left**]
- `(MIN_EXTENT, MAX_EXTENT)` [**upper left**]

- (MAX_EXTENT, MIN_EXTENT) [**lower right**]
- (MAX_EXTENT, MAX_EXTENT) [**upper right**]

Consider a Region instance consisting of three extents associated with the routing space. Each extent is itself a two-dimensional box characterized by the points

- (L_{xe}, L_{ye}) [**lower left**]
- (L_{xe}, U_{ye}) [**upper left**]
- (U_{xe}, L_{ye}) [**lower right**]
- (U_{xe}, U_{ye}) [**upper right**]

L and *U* denote lower or upper bound, *x* and *y* denote the two dimensions in the space, and *e* denotes the extent number within the region object. The subspace represented by this particular region is the union of the three boxes defined by the three extents.

Each extent in the region has an associated "notification box" that is at least as big as the box defined by the extent's ranges. The notification box associated with an extent is characterized by the four points

- ($L_{xe} - T_x$, $L_{ye} - T_y$) [**lower left**]
- ($L_{xe} - T_x$, $U_{ye} + T_y$) [**upper left**]
- ($U_{xe} + T_x$, $L_{ye} - T_y$) [**lower right**]
- ($U_{xe} + T_x$, $U_{ye} + T_y$) [**upper right**]

T represents the threshold distance for a given dimension, such that

$T_d = P_d$ * (MAX_EXTENT – MIN_EXTENT)

P denotes the real number in the range [0,1] that is defined as the threshold percentage for a particular dimension by the RID file.

The union of the three notification boxes defines the subspace that is considered relevant when the region instance is used for subscription.

Note that the coordinate grids associated with routing spaces need not correspond to geographical locations (although this is certainly a popular application.) The semantics of dimensions and ranges are entirely defined by the federation. Creative use of routing spaces can be used to implement data filtering and partitioning based on radio frequencies, event priorities, or almost any other criteria.

METHODS

~Region

The destructor is implicitly invoked whenever an instance of the class is deallocated. The destructor frees any resources associated with the class instance.

getNumberOfExtents

This method returns the number of extents used to describe the region, as bound to the instance during its creation.

getRangeLowerBound

This method queries the lower bound in the specified dimension of the subspace described by the specified extent.

If the specified extent is not valid for the region instance (i.e., *theExtent* < *0* or *theExtent* ≥ *getNumberOfExtents()*) or if the specified dimension is not valid in the context of the region's routing space, the ArrayIndexOutOfBounds exception is thrown.

getRangeLowerBoundNotificationLimit

This method queries the lower bound in the specified dimension of the subspace within which incoming events are considered "relevant" based on the current ranges comprising the specified extent.

An incoming event is considered relevant to a federate if the region associated with the event intersects any one of the subspaces defined by the notification limits of any extents of any subscribed region.

The notification limits associated with a region instance are based on the most recent version of the instance committed to the RTI using notifiyAboutRegionModification(). Changes made to extents of an instance after the most recent invocation of this service will not be reflected in the notification limits.

getRangeUpperBound

This method queries the upper bound in the specified dimension of the subspace descibed by the specified extent.

If the specified extent is not valid for the region instance (i.e., *theExtent* < *0* or *theExtent* ≥ *getNumberOfExtents()*) or if the specified dimension is not valid in the context of the region's routing space, the ArrayIndexOutOfBounds exception is thrown.

getRangeUpperBoundNotificationLimit

This method queries the upper bound in the specified dimension of the subspace within which incoming events are considered "relevant" based on the current ranges comprising the specified extent.

An incoming event is considered relevant to a federate if the region associated with the event intersects any one of the subspaces defined by the notification limits of any extents of any subscribed region.

The notification limits associated with a region instance are based on the most recent version of the instance committed to the RTI using notifiyAboutRegionModification(). Changes made to extents of an instance subsequent to the most recent invocation of this service will not be reflected in the notification limits.

getSpaceHandle

This method returns a handle to the routing space of which the region is a subset, as bound to the instance during its creation.

setRangeLowerBound

This method sets the lower bound in the specified dimension of the subspace described by the specified extent. The value of the lower bound should be no less than MIN_EXTENT and no greater than the upper bound of the specified dimension for the specified extent.

If the specified extent is not valid for the region instance (i.e., *theExtent* < *0* or *theExtent* ≥ *getNumberOfExtents()*) or if the specified dimension is not valid in the context of the region's routing space, the ArrayIndexOutOfBounds exception is thrown.

setRangeUpperBound

This method sets the upper bound in the specified dimension of the subspace described by the specified extent. The value of the upper bound should be no greater than MAX_EXTENT and no less than the lower bound of the specified dimension

for the specified extent.

If the specified extent is not valid for the region instance (i.e., *theExtent* < *0* or *theExtent* ≥ *getNumberOfExtents()*) or if the specified dimension is not valid in the context of the region's routing space, the `ArrayIndexOutOfBounds` exception is thrown.

SEE ALSO

Pound-Defined Constants
MAX_EXTENT
MIN_EXTENT
RTI::RTIambassador
createRegion()
deleteRegion()
getDimensionHandle()
getDimensionName()
getSpaceHandle()
getSpaceName()
notifyAboutRegionModification()

C.2 Supporting Types

C.2.1 Enumerated Types

RTI 1.3-NG

ABSTRACT

The RTI defines several enumerated types. Enumerated values are all-upper-case, with underscores used to separate words.

RTI::BOOLEAN

To support C++ compilers that do not have a built-in `bool` primitive, the RTI defines its own Boolean type.

RTI_TRUE

This value represents a positive (true) condition.

RTI_FALSE

This value represents a negative (false) condition.

RTI::FederateStateType

This enumeration is used by the Management Object Model to communicate the state of a federate or federation.

RESTORING

The federate or federation is in the process of restoring.

RUNNING

The federate or federation is in the normal state.

SAVING

The federate or federation is in the process of saving.

RTI::ObjectState

This enumeration is used by the Management Object Model to communicate the status of an object instance for an LRC.

DELETED

The object instance is in the LRC's database, but it is marked for deletion.

HOLDING_TOKENS

The object is not known to the federate, but one or more attribute-instance ownership tokens for the instance are being managed by the local LRC.

KNOWN_TO_FEDERATE

The object is known to the federate. Some attribute-instance ownership tokens for the instance may or may not be owned by the federate or managed by the local LRC.

RTI::ResignAction

This enumeration is used to instruct the `ResignFederationExecution()` service as to how locally owned attribute-instances should be resolved.

DELETE_OBJECTS

Any objects for which the federate holds the privilege to delete will be deleted. Any other attribute-instances owned by the federate will be lost to the federation.

DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES

The LRC first deletes any objects for which the federate holds the privilege to delete, then unconditionally divests ownership of any remaining attribute-instances owned by the federate.

NO_ACTION

No action is taken. All attribute-instances owned by the federate will be lost to the federation.

RELEASE_ATTRIBUTES

All attribute-instances owned by the federate are unconditionally divested, including any instances of the special "privilege to delete" attribute.

RTI::TimeManagerStateType

This enumeration is used by the Management Object Model to communicate whether or not a federate is advancing time at a given instant.

IDLE

There is not currently a time-advancement service in progress for the federate.

TIME_ADVANCING

There is a time-advancement service in progress for the federate.

RTI 1.0 MIGRATION NOTES

- The `OrderType` and `TransportType` enumeration's no longer exist in RTI 1.3; see the `getTransportationHandle()` and `getOrderingHandle()` services.
- The `OwnershipDivestitureCondition` enumeration no longer exists; there are now two separate services corresponding to the two values of this enumeration.
- `ObjectRemovalReason` no longer exists; the federate is not informed of the reason an object is being removed.
- `TokenState` no longer exists; this information is no longer made available to the federation.
- Many values have been eliminated from the `FederateStateType` enumeration; pause/resume no longer exists and save/restore-pending is no longer meaningful under the 1.3 save/restore mechanism.
- `FederationStateType` no longer exists; pause/restore has been replaced by synchronization points.
- `TimeManagerStateType` has been reduced to only two enumerated valued.

C.1.1 EXCEPTIONS

RTI 1.3-NG

ABSTRACT

Exceptions are the mechanism used by the RTI to communicate the fact that a method or service has failed to complete successfully.

DESCRIPTION

A number of different types of exceptions are thrown by the RTI; each type is represented as a separate C++ class to facilitate the use of different exception handlers to catch different kinds of exceptions. See the section on the `Exception` class for the structure of the thrown exceptions. This section presents a brief general description of each type of exception; for notes on the use of an exception in the context of a particular method or service, see the section on the method or service.

EXCEPTIONS THROWN ONLY BY THE RTI

ArrayIndexOutOfBounds

An invalid argument was provided to a method of one of the RTI utility data-structure classes.

AsynchronousDeliveryAlreadyDisabled

An attempt to disable asynchronous delivery of receive-ordered events was made when asynchronous delivery was already disabled.

AsynchronousDeliveryAlreadyEnabled

An attempt to enable asynchronous delivery of receive-ordered events was made when asynchronous delivery was already enabled.

AttributeAlreadyBeingAcquired

A request was made to acquire an attribute-instance for which the local federate already had an outstanding acquisition request.

AttributeAlreadyBeingDivested

A request was made to divest an attribute-instance for which the local federate already had an outstanding divestiture request.

AttributeNotDefined

An attribute handle was used that was invalid in the specified object-class or object-instance context.

ConcurrentAccessAttempted

An attempt was made to invoke a non-reentrant service while an invocation of a non-reentrant service is still in progress; this occurs when RTI ambassador services are invoked from federate ambassador callbacks or from multiple threads simultaneously.

CouldNotOpenFED

The user provided FED file was not found in the specified location.

DeletePrivilegeNotHeld

An attempt was made to delete an object for which the local federate does not own the *privilegeToDelete* attribute-instance.

DimensionNotDefined

The specified name was not a dimension of the specified routing space.

EnableTimeConstrainedPending

An attempt was made to enable time constraint or advance time while a request to enable time constraint was still in progress.

EnableTimeRegulationPending

An attempt was made to enable time regulation or advance time while a request to enable time regulation was still in progress.

ErrorReadingFED

A file was found in the specified FED file location, but it was not in the correct format.

FederateAlreadyExecutionMember

An attempt was made to join a federation execution when the RTI ambassador was already associated with a federation.

FederateLoggingServiceCalls

This exception is not thrown.

FederateNotExecutionMember

A service invocation that is only valid in the context of a federation was made when the RTI ambassador was not associated with any federation.

FederatesCurrentlyJoined

An attempt was made to destroy a federation execution to which there are still joined federates.

FederateWasNotAskedToReleaseAttribute

The federate attempted to respond to an attribute-instance release request when no release request was outstanding for the attribute-instance.

FederationExecutionAlreadyExists

An attempt was made to create a federation execution that already existed.

FederationExecutionDoesNotExist

An attempt was made to join a federation execution that did not exist.

FederationTimeAlreadyPassed

An attempt was made to schedule a save or advance federate logical time to a logical time that was in the federation's past.

HandleValuePairMaximumExceeded

This exception is not thrown by RTI 1.3.

InteractionClassNotDefined

An invalid interaction-class handle was used as an argument to a service invocation.

InteractionClassNotSubscribed

A service or callback was invoked that expected its subject to be an interaction class that was not currently subscribed by the local federate, but the interaction class was currently subscribed by the federate.

InteractionParameterNotDefined

A parameter handle was used that was invalid in specified interaction-class context.

InvalidExtents

The extents associated with a region instance were invalid, i.e. a lower bound was greater than an upper bound, or an extent boundary was outside the range defined by `[MIN_EXTENT,MAX_EXTENT]`.

InvalidHandleValuePairSetContext

A method of a pair set that is only valid for pair-sets created by the LRC to communicate an update or interaction to the federate was invoked on a set that was not an argument to a reflection or receipt.

InvalidLookahead

An attempt was made to turn regulation on or modify the federate's lookahead was made with an invalid logical time argument.

InvalidOrderingHandle

An ordering handle was not recognized as being a valid handle as returned by `getOrderingHandle()`.

InvalidRegionContext

An attempt was made to delete a region that was still associated with attribute-instances.

InvalidResignAction

A resign-action argument was not recognized as being a valid enumerated value of the `ResignAction` type.

InvalidRetractionHandle

An attempt was made to retract an event that was not sent by the local federate, or no longer in the history buffer of the local LRC.

InvalidTransportationHandle

An ordering handle was not recognized as being a valid handle as returned by `getTransportationHandle()`.

MemoryExhausted

Insufficient memory was available to fulfill an allocation request.

NameNotFound

The specified symbolic (string) name was not valid in the context used.

ObjectAlreadyRegistered

An attempt was made to register an object instance with a symbolic (string) name that was not unique to the federation at that point.

ObjectClassNotDefined

An invalid object-class handle was used.

ObjectClassNotSubscribed

A service or callback was invoked that expected its subject to be an object class that was currently subscribed by the local federate, but the class was not subscribed by the local federate.

RegionNotKnown

A service was invoked with a `Region` object that was not recognized as a valid instance as created by the `getRegion()` service.

RestoreInProgress

A service invocation that is not allowed during a restoration was made while a federation restoration was in progress.

RestoreNotRequested

The federate reported a completed restoration attempt when it had not been asked to restore.

RTIinternalError

An error internal to the RTI occurred; consult the federate's log file for details.

SaveInProgress

A service invocation that is not allowed during a save was made while a federation save was in progress.

SaveNotInitiated

The federate reported a completed save attempt when it had not been asked to save.

SpaceNotDefined

An invalid space handle was used as an argument to an RTI service.

SynchronizationPointLabelWasNotAnnounced

The federate reported the achievement of a synchronization point for while there was not an outstanding synchronization request.

TimeAdvanceAlreadyInProgress

An invocation of a time-advancement service before the previous time-advancement service had culminated in a time-advance grant.

TimeConstrainedAlreadyEnabled

The federate attempted to enable time constraint when time constraint was already enabled.

TimeConstrainedWasNotEnabled

The federate attempted to disable time constraint when time constraint was already disabled.

TimeRegulationAlreadyEnabled

The federate attempted to enable time regulation when time regulation was already enabled.

TimeRegulationWasNotEnabled

The federate attempted to disable time regulation when time regulation was already disabled.

ValueCountExceeded

An attempt was made to allocate a set with greater capacity than is allowed, or an attempt to insert an item into a set would result in its capacity being exceeded.

ValueLengthExceeded

Insufficient memory was available to fulfill a request to add an attribute or parameter value to a handle-value pair set.

EXCEPTIONS THROWN BY THE RTI OR THE FEDERATE

AttributeAlreadyOwned

A service or callback was invoked that expected its subject to be an attribute-instance that is not currently owned by the local federate, but the local federate does currently own the attribute-instance.

AttributeAcquisitionWasNotRequested

A service or callback was invoked that expected its subject to be an attribute-instance for which an acquisition request was outstanding, but there was no outstanding acquisition request by the local federate.

AttributeDivestitureWasNotRequested

A service or callback was invoked that expected its subject to be an attribute-instance for which a divestiture request was outstanding, but there was no outstanding divestiture request by the local federate.

AttributeNotOwned

A service or callback was invoked that expected its subject to be an attribute-instance that is currently owned by the local federate, but the local federate does not currently own the attribute-instance.

AttributeNotPublished

A service or callback was invoked that expected its subject to be an class-attribute that the local federate is currently publishing, but the federate is not currently publishing the attribute.

FederateOwnsAttributes

A service or callback was invoked that expected its subject to be an attribute-instance that is not currently owned by the local federate, but the local federate does currently own the attribute-instance.

InteractionClassNotPublished

A service or callback was invoked that expected its subject to be an interaction class that was currently published by the local federate, but the interaction class was not currently published by the federate.

InvalidFederationTime

A logical time argument to a service or callback was not a valid point on the federation logical time axis.

ObjectClassNotPublished

A service or callback was invoked that expected its subject to be an object class that was currently published by the local federate, but the class was not published by the local federate.

ObjectNotKnown

An object instance handle was used as an argument to a service or callback that did not correspond to an object instance known to the LRC or the federate, respectively.

SpecifiedSaveLabelDoesNotExist

The federate was asked to restore to a save label that did not correspond to a saved federate state. (This exception is not thrown by the RTI ambassador.)

EXCEPTIONS THROWN ONLY BY THE FEDERATE

AttributeAcquisitionWasNotCanceled

A callback to confirm the cancellation of an attribute acquisition was made for an attribute-instance that was not the subject of a previous acquisition cancellation.

AttributeNotKnown

An attribute handle was used that was invalid in the specified object-class or object-instance context.

CouldNotDiscover

The federate failed to discover an object for some reason other than an unrecognized object-class handle.

CouldNotRestore

This exception should not be thrown by the federate; use `federateRestoreNotComplete()` instead.

EnableTimeConstrainedWasNotPending

The federate received a callback advising it that time constraint had been enabled, but it had not requested time constraint to be enabled.

EnableTimeRegulationWasNotPending

The federate received a callback advising it that time constraint had been enabled, but it had not requested time constraint to be enabled.

EventNotKnown

A request was made to retract an event that was not known to the federate.

FederateInternalError

An error internal to the federate prevented it from successfully processing a federate ambassador callback.

InteractionClassNotKnown

An invalid interaction-class handle was used as an argument to a service invocation.

InteractionParameterNotKnown

A parameter handle was used that was invalid in specified interaction-class context.

ObjectClassNotKnown

An invalid object-class handle was used as an argument to a service or callback.

TimeAdvanceWasNotInProgress

The federate was informed of a time-advance grant when there was not any time-advancement service in progress.

UnableToPerformSave

This exception should not be thrown by the federate; use `federateSaveNotComplete()` instead.

C.2.2 Factory Classes

RTI 1.3-NG

ABSTRACT

Many RTI utility classes are abstract and cannot be directly instantiated. Factories are simple classes, typically consisting of a single static method, which are used to create instances of utility classes.

SYNOPSIS

```
#include <RTI.hh>

class RTI::AttributeSetFactory {
public:
   static
   RTI::AttributeHandleValuePairSet*
       create(RTI::ULong count)
           throw (
               RTI::MemoryExhausted,
               RTI::ValueCountExceeded
       );
};

class RTI::AttributeHandleSetFactory {
public:
   static
   RTI::AttributeHandleSet*
   create(RTI::ULong count)
       throw(
           RTI::MemoryExhausted,
           RTI::ValueCountExceeded
       );
};

class RTI::FederateHandleSetFactory {
public:
   static
   RTI::FederateHandleSet*
       create(RTI::ULong count)
           throw (
               RTI::MemoryExhausted,
               RTI::ValueCountExceeded
           );
};

class RTI::ParameterSetFactory {
public:
   static
   RTI::ParameterHandleValuePairSet*
       create(RTI::ULong count)
           throw (
               RTI::MemoryExhausted,
               RTI::ValueCountExceeded,
               RTI::HandleValuePairMaximumExceeded
           );
};

class RTI::FedTimeFactory {
public:
   static
   RTI::FedTime*
       makeZero( )
           throw (
               MemoryExhausted
           );

   static
   RTI::FedTime*
       decode(const char *buf)
           throw (
               RTI::MemoryExhausted
           );
};
```

DESCRIPTIONS

All factories throw the `MemoryExhausted` exception if there is not sufficient memory to fulfill the allocation request.

AttributeHandleSetFactory

The `create()` method returns a pointer to a newly allocated `AttributeHandleSet` instance. The *count* argument is not used; all sets may contain any and all handles from *0* to *MAX_ATTRIBUTES_PER_CLASS – 1*. The new set is initially empty.

The `ValueCountExceeded` exception is not thrown.

AttributeSetFactory

The `create()` method returns a pointer to a newly allocated `AttributeHandleValuePairSet` instance. The *count* argument specifies a limit on the number of handle-value pairs that may be inserted into the set. The new set initially contains no handle-value pairs.

The `ValueCountExceeded` exception is thrown if the specified limit is greater than the `MAX_ATTRIBUTES_PER_CLASS` constant.

FederateHandleSetFactory

The `create()` method returns a pointer to a newly allocated `FederateHandleSet` instance. The *count* argument specifies a limit on the number of handles that may be inserted into the set. The new set is initially empty.

The `ValueCountExceeded` exception is thrown if the specified limit is greater than the `MAX_ATTRIBUTES_PER_CLASS` constant. (It should only ever be necessary to have `MAX_FEDERATE` handles in the set.)

FedTimeFactory

The `makeZero()` method returns a pointer to a new `FedTime` instance initialized to logical time zero (the lower bound on the federation logical time axis.)

The `decode()` method returns a pointer to a new `FedTime` instance initialized to an arbitrary value. The *buf* argument should be a pointer to a non-negative C++ `char*` with which to initialize the new instance.

ParameterSetFactory

The `create()` method returns a pointer to a newly allocated `ParameterHandleValuePairSet` instance. The *count* argument specifies a limit on the number of handle-value pairs that may be inserted into the set. The new set initially contains no handle-value pairs.

The `ValueCountExceeded` exception is thrown if the specified limit is greater than the `MAX_PARAMETERS_PER_CLASS` constant.

C.2.3 Pound-Defined Constants

RTI 1.3-NG

ABSTRACT

Pound-defined constants are used for RTI parameters that are not configurable via the RID file. By convention, pound-defined constants are all upper-case with underscores separating the words.

Identifier	Type	Value	Description
MAX_FEDERATION	unsigned short	32	the maximum number of simultaneous federations supported
MAX_FEDERATE	unsigned short	32	the maximum number of federates per federation supported
MAX_NAME_LENGTH	size_t	64	the maximum length of federation and federate names (including null-terminator)
MAX_SPACES	unsigned short	10	the maximum number of spaces per federation (including the default space) supported
MAX_OBJECT_CLASSES	unsigned short	200	the maximum number of object classes allowed in the FED (including MOM and internal RTI classes)
MAX_INTERACTION_CLASSES	unsigned short	200	the maximum number of interaction classes allowed in the FED (including MOM and internal RTI classes)
MAX_ATTRIBUTES_PER_CLASS	unsigned short	200	the maximum number of attributes allowed for an object class (includes attributes inherited from superclasses)
MAX_PARAMETERS_PER_CLASS	unsigned short	200	the maximum number of parameters allowed for an interaction class (includes parameters inherited from superclasses)
MAX_DIMENSIONS_PER_SPACE	unsigned short	10	the maximum number of dimensions allowed for a single routing space
DEFAULT_SPACE_NAME	string	“defaultSpace”	the name of the default routing space implicitly defined for all federations (note: space names are case-insensitive)
RTI_VERSION	string	“1.3R11”	string identifier representing the version of the RTI software
MAX_EXTENT	unsigned long	0xc0000000	the upper bound of the range of values associated with a single dimension of a routing space
MIN_EXTENT	unsigned long	0x40000000	the lower bound of the range of values associated with a single dimension of a routing space

C.2.4 EventRetractionHandle

RTI 1.3-NG

ABSTRACT

Event-retraction handles uniquely identify time-stamped events in the federation.

SYNOPSIS

```
#include <RTI.hh>

struct EventRetractionHandle_s {
      RTI::UniqueID        theSerialNumber;
      RTI::FederateHandle  sendingFederate;
};

typedef struct  EventRetractionHandle_s
                EventRetractionHandle;
```

DESCRIPTION

Event-retraction handles uniquely identify time-stamped events in the federation. They are used by the RTI for purposes of retracting events using the `retract()` and `requestRetraction()` services. They may also be used by the federation in any circumstances where time-stamped events need to be referred to individually.

Note that the association of a time-stamp (and hence a retraction handle) with an event does not imply that the event is sent or delivered in time-stamp order.

MEMBERS

sendingFederate

the federate handle of the federate that produced the event; such handles are guaranteed to be unique at any particular instant in the federation but not necessarily over the entire lifetime of the federation

theSerialNumber

the sequence number of the event relative to other events generated by the same federate; the sequence number is incremented by one for each event sent by the federate and will therefore be unique to the federate (subject to the limitations of the integer representation)

SEE ALSO

RTI::RTIambassador::
retract()
RTI::FederateAmbassador::
requestRetraction()

C.2.5 TYPEdefs

RTI 1.3-NG

ABSTRACT

The RTI uses typedefs to create descriptive aliases for primitive C++ types. The C++ primitive name may be used interchangeably with typedef'd names; however, using the descriptive names may make code more readable

Old Type	New Type	Structure	Usage
unsigned short	RTI::UShort	2-byte integer in range [0,65535]	platform-independent reference to 2-byte unsigned integer
short	RTI::Short	2-byte integer in range [-32768,32767]	platform-independent reference to 2-byte signed integer
unsigned long[1]	RTI::ULong	4-byte integer in range [0,4294967295]	platform-independent reference to 4-byte unsigned integer
long[1]	RTI::Long	4-byte integer in range [-2147483648,2147483647]	platform-independent reference to 4-byte signed integer
double	RTI::Double	8-byte floating point in range [-1.7976931348623157E+308, 1.7976931348623157E+308][2]	platform-independent reference to 8-byte floating point number
float	RTI::Float	4-byte floating point in range [-3.402823466E+38, 3.402823466E+38][2]	platform-independent reference to 4-byte floating point number
RTI::FederateAmbassador*	RTI::FederateAmbassadorPtr	4-byte address of `RTI::FederateAmbassador`[3]	shorthand for pointer
RTI::Long	RTI::SpaceHandle	4-byte integer in range [-2147483648,2147483647]	used to refer to RTI-assigned values representing federation routing spaces
RTI:: ULong	RTI::ObjectClassHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-assigned values representing federation object classes
RTI::ULong	RTI::InteractionClassHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-assigned values representing federation interaction classes
RTI::ULong	RTI::ExtentIndex	4-byte integer in range [0,4294967295]	used to refer to extents comprising a `Region` instance
RTI::ULong	RTI::Handle	4-byte integer in range [0,4294967295]	shorthandle for `Handle`
RTI::Handle	RTI::AttributeHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-assigned values representing object-class attributes
RTI::Handle	RTI::ParameterHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-assigned values representing interaction-class parameters
RTI::Handle	RTI::ObjectHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-assigned object instances in the federation
RTI::Handle	RTI::DimensionHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-defined values representing dimensions of a federation routing space
RTI::ULong	RTI::FederateHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-defined values representing active federates
RTI::Handle	RTI::TransportationHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-defined values representing categories of transportation service
RTI::TransportationHandle	RTI::TransportType	4-byte integer in range [0,4294967295]	used to refer to RTI-defined values representing categories of transportation service
RTI::Handle	RTI::OrderingHandle	4-byte integer in range [0,4294967295]	used to refer to RTI-defined values representing categories of ordering service
RTI::OrderingHandle	RTI::OrderType	4-byte integer in range [0,4294967295]	used to refer to RTI-defined values representing categories of ordering service
RTI::ULong	RTI::FederateID	4-byte integer in range [0,4294967295]	not used in RTI 1.3
RTI::ULong	RTI::UniqueID	4-byte integer in range [0,4294967295]	used to refer to unique serial numbers used to retract time-stamped events

[1] On platforms with 8-byte longs, this will be an `int` to keep the number of bytes consistent across platforms.
[2] The values listed are for the Sparc Solaris platform. Other platforms may differ.
[3] On some platforms, an address may be 8 bytes.

Old Type	New Type	Structure	Usage
RTI::Double	RTI::TickTime	8-byte floating point in range [-1.7976931348623157E+308, 1.7976931348623157E+308] [2]	used to refer to the amount of time to spend in tick (in seconds)
RTI::ULong	RTI::RegionToken	4-byte integer in range [0,4294967295]	used to refer to `Region` instance serial numbers used for marshaling