normativeTypes (C++)

Apr 15, 2020

Contents

CHAPTER 1

normativeTypesCPP Reference

1.1 Release 5.3.0 - September 2019

1.2 Abstract

The EPICS 7 PVA modules provide efficient storage, access, and communication, of memory resident structured data. The PVA Normative Types are a collection of structured data types that can be used by the application level of EPICS 7 network endpoints, to interoperably exchange scientific data. normativeTypesCPP is the C++ implementation. It is one part of the set of related products in the EPICS 7 control system toolkit.

1.3 Status of this Document

This is the 05 September 2019 version for the 5.3.0 release of the C++ implementation of Normative Types.

RELEASE_NOTES.md provides changes since the last release. TODO.md describes things to do before the next release.

1.4 Introduction

This manual assumes that the reader is familiar with the material in the pvDataCPP Documentation

At present the following Normative Types are implemented:

- **NTScalar**
- **NTScalarArray**

- **NTEnum**
- **NTMatrix**
- **NTURI**
- **NTNameValue**
- **NTTable**
- **NTAttribute**
- **NTMultiChannel**
- **NTNDArray**
- **NTContinuum**
- **NTHistogram**
- **NTAggregate**

There is also additional support for NTAttributes which are extended as required by NTNDArray.

Thus normativeTypesCPP implements fully the March 16 2015 version the Normative Types Specification.

Each Normative Type consists of a set of mandatory fields, a set of optional fields, and any arbitrary number of additional fields. The mandatory and optional fields are meant for use by standard tools such as Display Managers and Alarm Handlers. The additional fields are for specialized tools.

A helper class NTField is provided to enforce the proper implementation of property fields as defined by pvData. A property field is normally associated with a field that has the name "value". The property fields currently used are alarm, timeStamp, display, control, and alarmLimit. In addition pvData defines a standard structure for a field that represents enumerated values. NTField has methods associated with each of these.

An include file "nt.h" includes all the other header files that are provided by ntCPP.

The next section gives an overview of the library. The following sections describe NTField and then the Normative Type classes, starting with features common to all classes, then support in the Normative Type classes for property fields. Finally the classes for each Normative Type are described.

1.5 Overview (Informative)

1.5.1 Normative Type classes

For each Normative Type there is a corresponding wrapper class that provides a convenient API for manipulating a `PVStructure` conformant to the given Normative Type.

The class names match the names of the Normative Types, so the wrapper class for NTScalar is `NTScalar`.

Each wrapper class provides functions for identifying the Normative Type of a `Structure` or `PVStructure` and validating conformance.

The wrapper classes can create wrappers around an existing `PVStructure`. They can also, through a builder class, create a conformant `Structure` or `PVStructure` or create a wrapper around a new, conformant `PVStructure`. Builders will create all required fields and can also create optional and additional fields and handle any choices in the definition of the structure, such as the `ScalarType` of a NTScalar.

The builder for each class is named in an obvious way, so for example the builder for `NTScalar` is `NTScalarBuilder`.

As usual in EPICS 7 C++ libraries, extensive use is made of shared pointers and these can be dealt with via typedefs. So for NTScalar we have `NTScalarPtr` and `NTScalarBuilderPtr`.

In the following examples it is assumed that the namespaces `epics::pvData` , `epics::nt` and `std` are used.

1.5.2 Creating Normative Types

Creating a builder class

Each Normative Type wrapper has a static `createBuilder()` function which creates a builder for the Normative Type. The following creates a builder class for an NTScalar:

```
NTScalarBuilderPtr builder = NTScalar::createBuilder();
```

Creating Structures, PVStructures and wrappers

Each Normative Type builder has `createStructure()` and `createPVStructure()` functions which respectively create a `Structure` or `PVStructure` conformant to the Normative Type. The builders also each have a `create()` function which creates a new conformant `PVStructure` and returns a wrapper around it.

The following creates a `Structure`, a `PVStructure` and a wrapper class instance for NTEnum:

```
StructureConstPtr structure   = NTEnum::createBuilder()->createStructure();
PVStructurePtr    pvStructure = NTEnum::createBuilder()->createPVStructure();
NTEnumPtr         wrapper     = NTEnum::createBuilder()->create();
```

The structures created by the above functions will have all required fields of the Normative Type. Unless requested to do so the builder will not include any optional or additional fields. The mechanism for doing this is described below.

The above three methods cause a builder to be reset, so any additional information supplied, such adding optional or additional fields, is lost at this point.

Types requiring information before construction

Some Normative Types require information to be supplied before a conformant `Structure` or `PVStructure` can be constructed. Good examples are the types NTScalar and NTScalarArray which require the `ScalarType` to be supplied:

```
NTScalarPtr      scalar = NTScalar::createBuilder()->value(pvDouble)->create();
NTScalarArrayPtr array  = NTScalarArray::createBuilder()->value(pvString)->create();
```

This produces wrappers around the following `PVStructures`:

```
epics:nt/NTScalar:1.0
    double value 0

epics:nt/NTScalarArray:1.0
    string[] value []
```

In the above cases not specifying a `ScalarType` causes an exception (`std::runtime_error`) to be thrown.

The same is true for NTNameValue and NTHistogram.

See individual types for more information.

Optional fields

Each builder has functions for adding optional fields to the constructed structure. Each returns the builder so that methods can be chained.

The following will produce a wrapper for a NTScalar with `descriptor`, `alarm`, `timeStamp`, `display` and `control` fields:

```
NTScalarPtr scalar = NTScalar::createBuilder()->
    value(pvDouble)->
    addDescriptor()->
    addAlarm()->
    addTimeStamp()->
    addDisplay()->
    addControl()->
    create();
```

The names of the add methods are in each case "add" plus the name of the field (with case suitably adjusted). So `addAlarm()` adds the `alarm` field.

The order of the fields in the created structure is that laid out in the Normative Types specification, not the order that the functions are called.

The optional fields selected in the builder are reset by calling `create()`, `createStructure()` or `createPVStructure()`.

Additional fields

Each builder has an `add()` function for adding additional fields to the constructed structure. For example

```
PVStructurePtr pvStructure = NTScalar::createBuilder()->
    value(pvDouble)->
    add("tags", getFieldCreate()->createScalarArray(pvString))->
    createPVStructure();
```

produces

```
epics:nt/NTScalar:1.0
    double value 0
    string[] tags []
```

Again, each function returns the builder so that methods can be chained

Currently the second argument to `add` can only be a `Field` (a `ScalarType` , for example, is not possible).

The order of the additional fields is the order that the `add()` functions are called, but, as required by the Normative Types specification, the additional fields will follow the required fields and any optional fields, regardless of whether an `add` function call comes before or after a call to add an optional field.

The additional fields selected in the builder are reset by calling `create()`, `createStructure()` or `createPVStructure()`.

Other type-dependent builder options

Some types have additional builder functions:

- As mentioned `NTScalar`, `NTScalarArray`, `NTNameValue` and `NTHistogram` require the `ScalarType` of their `value` fields to be specified through their builder's `value()` function.
- `Similarly NTScalarMultiChannel` has a `value` field whose `ScalarType` is set via a `value()` function. (It however defaults to a "double".)
- NTUnion has a `value()` function which set the union type of its union `value` field. NTMultiChannel has a `value()` function which sets the type of its union array `value` field. (Default is a variant union in each case.)
- NTTable has an `addColumn()` function which adds a column to the table.
- NTURI has `addQueryString()`, `addQueryDouble()` and `addQueryInt()` functions which add fields to the query field.

These are all reset by calling `create()`, `createStructure()` or `createPVStructure()`.

They are described in the corresponding section for each type.

1.5.3 Checking and Wrapping Existing Structures

[In the following `structure` is a `StructureConstPtr`, `pvStructure` is a `PVStructurePtr`.]

Checking for compatible type ID

Each Normative Type wrapper has a static `is_a()` function which looks at the type ID and tests whether this is consistent with the given Normative Type.

The following tests whether `structure` reports to be an NTScalar:

```
if (!NTScalar::is_a(structure))
    cout << "Structure's ID does not report to be an NTScalar" << endl;
```

Similarly for `pvStructure`:

```
if (!NTScalar::is_a(pvStructure))
    cout << "PVStructure's ID does not report to be an NTScalar" << endl;
```

Checking for compatible introspection type

Each Normative Type wrapper has a static `isCompatible()` function which tests for compatibility based on introspection data only.

The following tests whether `structure` is compatible with the definition of NTEnum:

```
if (!NTEnum::isCompatible(structure))
    cout << "Structure is not compatible with NTEnum" << endl;
```

Similarly for `pvStructure`:

```
if (!NTEnum::is_a(pvStructure))
    cout << "PVStructure is not compatible with NTEnum" << endl;
```

Wrapping a PVStructure (without checks)

Each Normative Type wrapper has a static `wrapUnsafe()` function which creates a wrapper around an existing `PVStructure`.

The following creates an NTScalarArray wrapper around an existing `pvStructure`:

```
NTScalarArrayPtr array = NTScalarArray::wrapUnsafe(pvStructure);
```

If `isCompatible()` returns true, the Normative Type wrapper functions may be safely called.

Wrapping a PVStructure (with checks)

Each Normative Type wrapper also has a static `wrap()` function which checks checks compatibility. It is equivalent to calling `isCompatible()` and returning `wrapUnsafe()` if `true` or a null pointer if `false`:

```
NTScalarArrayPtr array = NTScalarArray::wrap(pvStructure);
if (!array.get())
    cout << "PVStructure is not compatible with NTScalarArray." << endl;
```

Checking validity of a PVStructure

Each Normative Types wrapper's `isCompatible()` function only checks the introspection data.

To perform any checks on the `PVStructure`'s value data use the wrapper's (non-static) `isValid()` function.

For example

```
NTTablePtr table = NTTable::wrap(pvStructure);
if (table.get() && table->isValid())
    cout << "Table is valid" << endl;
```

will check that a `PVStructure` is both compatible with NTTable and that it is valid in terms of its value data. In the case of NTTable the checks are that the columns are of equal length and the number of labels matches the number of columns.

For many types there is no appropriate check to be made on the value data. The function just returns true in this case.

1.5.4 Normative Type Wrapper Functions

Getting PVStructures

Each Normative Type wrapper has a `getPVStructure()` function which returns the wrapped `PVStructure`.

```
NTScalarPtr scalar = NTScalar::createBuilder()->value(pvDouble)->create();
PVStructurePtr pvStructure = scalar->getPVStructure();
```

Accessing required and optional fields

Each Normative Type wrapper has offers a slightly more convenient API for accessing the fields of the wrapped `PVStructure`.

The API is dependent on the wrapper class, but typically each wrapper has an accessor function for most, if not all, required or optional Normative Type fields, and typically the names of these functions follow the pattern "get" + field name (with case adjusted). So to get the `value` field the function `getValue()` is used.

```
NTAggregatePtr aggregate = NTAggregate::createBuilder()->
    addDispersion()->
    addFirst()->
    addLast()->
    addMax()->
    addMin()->
    create();
aggregate->getValue()->put(2.5);
aggregate->getN()->put(100);
aggregate->getDispersion()->put(0.5);
aggregate->getFirst()->put(2.1);
aggregate->getLast()->put(3.1);
aggregate->getMax()->put(3.7);
aggregate->getMin()->put(1.1);
```

In some cases a field of a Normative Type can may be one of a variety of types, in which case a template function is often provided:

```
NTScalarPtr scalar = NTScalar::createBuilder()->
    value(pvDouble)-> create();
scalar->getValue<PVDouble>()->put(42);
```

See the corresponding section for each type for list of accessor functions.

1.5.5 Property Functions

Time stamps

Most Normative Types have an optional `timeStamp` field (NTURI is the exception). If a `PVStructure` conformant to a Normative Type has a `timeStamp` field, a `PVTimeStamp` can be attached and an `attachTimeStamp` function is provided to facilitate this:

```
NTScalarPtr scalar = NTScalar::createBuilder()->
    value(pvDouble)->addTimeStamp()->create();

PVTimeStamp pvTimeStamp;
scalar->attachTimeStamp(pvTimeStamp);

TimeStamp timeStamp;
timeStamp.getCurrent();
pvTimeStamp.set(timeStamp);
```

1.6 NTField

These are helper classes for creating standard fields for Normative Types. There is a single instance of this class, which is obtained via NTField::get().

```
class NTField{
public:
    static NTFieldPtr get();
    ~NTField() {}

    bool isEnumerated(FieldConstPtr const & field);
    bool isTimeStamp(FieldConstPtr const & field);
    bool isAlarm(FieldConstPtr const & field);
    bool isDisplay(FieldConstPtr const & field);
    bool isAlarmLimit(FieldConstPtr const & field);
    bool isControl(FieldConstPtr const & field);

    StructureConstPtr createEnumerated();
    StructureConstPtr createTimeStamp();
    StructureConstPtr createAlarm();
    StructureConstPtr createDisplay();
    StructureConstPtr createControl();

    StructureArrayConstPtr createEnumeratedArray();
    StructureArrayConstPtr createTimeStampArray();
    StructureArrayConstPtr createAlarmArray();
};
```

where

isEnumerated() Is the field an enumerated structure?

isTimeStamp() Is the field an timeStamp structure?

isAlarm() Is the field an alarm structure?

isDisplay() Is the field an display structure?

isAlarmLimit() Is the field an alarmLimit structure?

isControl() Is the field an control structure?

createEnumerated() Create an introspection interface for an enumerated structure.

createTimeStamp() Create an introspection interface for a timeStamp structure.

createAlarm() Create an introspection interface for an alarm structure.

createDisplay() Create an introspection interface for a display structure.

createControl() Create an introspection interface for a control structure.

createEnumeratedArray() Create an introspection interface for an structureArray of enumerated structures.

createTimeStampArray() Create an introspection interface for an structureArray of timeStamp structures.

createAlarmArray() Create an introspection interface for an structureArray of alarm structures.

1.7 Features common to all Normative Types

This section details features which are common to all Normative Type wrapper classes and their builders.

1.7.1 Organisation and Naming Conventions

The name of the corresponding wrapper class for each Normative Type matches the name of the type and the name of the builder class is the name of the type + `Builder`. So the wrapper class and builder for NTScalar are `NTScalar` and `NTScalarBuilder`. The builder classes are inside the namespace `detail`.

The header name is that of the Normative Type plus thr ".h" extension, with case suitably adjusted. So NTScalar is defined in "ntscalar.h".

Through the `POINTER_DEFINITIONS` macro typedefs `NTType::shared_pointer` and `detail::NTTypeBuilder::shared_pointer` are defined to the shared pointers to `NTType` and `NTTypeBuilder`, where NTType is the name of the Normative Type.

In turn the typedefs `NTTypePtr` and `NTTypeBuilderPtr` are also declared.

So for NTTScalar the typedefs `NTScalarPtr` and `NTScalarBuilderPtr` are declared for `NTScalar::shared_pointer` and `detail::NTScalarBuilder::shared_pointer`.

1.7.2 Features common to all Normative Type Builder classes

For a Normative Type NTType the builder class definition is equivalent to one of the form:

```
class NTType;
typedef std::tr1::shared_ptr<NTType> NTTypePtr;

namespace detail {
```

(continues on next page)

(continued from previous page)

```
class NTTypeBuilder
{
public:
    POINTER_DEFINITIONS(NTTypeBuilder);
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTTypeBuilderPtr create();
    shared_pointer add(
         string const & name,
         FieldConstPtr const & field);

// ... Remainder of class definition
}

typedef std::tr1::shared_ptr<detail::NTTypeBuilder>  NTTypeBuilderPtr;

}
```

where

createStructure() Creates an `Structure` for an NTType. Resets the builder.

createPVStructure() Create an `PVStructure` for an NTType. Resets the builder.

create() Creates an `PVStructure` for an NTType and creates an NTType wrapper class instance around it. Resets the builder.

add() Adds an additional field. Its name must not be that of a required field nor of an optional field (regardless of whether the optional field has been added). The order of the additional fields matches the order in which the calls of `add()` are made.

All builders include the functions to add the optional fields of the normative type. The order of fields in the final created structure is that laid out in the Normative Types specification, not the order that the functions are called.

The optional fields selected in the builder as well as the additional fields are reset by calling `create()`, `createStructure()` or `createPVStructure()`.

1.7.3 Features common to all Normative Type Wrapper classes

For a Normative Type NTType the wrapper class definition is equivalent to one of the form:

```
class NTType;
typedef std::tr1::shared_ptr<NTType> NTTypePtr;

class NTType
{
public:
    POINTER_DEFINITIONS(NTType);

    static const string URI;

    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);

    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
```

(continues on next page)

(continued from previous page)

```
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);

    bool isValid();

    static NTTypeBuilderPtr createBuilder();

    PVStructurePtr getPVStructure() const;
}
```

where

URI The type ID of any constructed structures. Also used in any compatibility checks.

is_a() Checks if the specified `Structure`/`PVStructure` reports compatibility with this version of NTType through its type ID, including checking version numbers. The return value does not depend on whether the structure is actually compatible.

isCompatible() Checks whether the supplied `Structure` or `PVStructure` is conformant with this version of NTType through the introspection interface.

wrapUnsafe() Creates an NTType wrapping the specified `PVStructure`, regardless of the latter's compatibility. No checks are made as to whether the specified `PVStructure` is compatible with NTScalar or is non-null.

wrap() Creates an NTType wrapping the specified `PVStructure` if the latter is compatible. Checks the supplied PVStructure is compatible with NTType and if so returns an NTType which wraps it, otherwise it returns null.

isValid() Returns whether the wrapped `PVStructure` is valid with respect to this version of NTType. Unlike `isCompatible()`, `isValid()` may perform checks on the value data as well as the introspection data.

getPVStructure() Returns the PVStructure that this instance wraps.

1.8 Normative Type Property Features

1.8.1 Normative Type support for descriptor fields

Most Normative Types have an optional `descriptor` field of the form

```
string descriptor
```

The corresponding Normative Type wrapper classes and their builders have support for this field:

Builder support

Each builder class for a Normative Type with a `descriptor` field has a function

```
shared_pointer addDescriptor();
```

where

addDescriptor() Adds the `descriptor` field to the structure returned by calling `create()`, `createStructure()` or `createPVStructure()`. Returns the the instance of the builder.

The effect of calling `addDescriptor()` is reset by a call of `create()`, `createStructure()` or `createPVStructure()`.

Normative Type class support

Each wrapper class for a Normative Type with a `descriptor` field has a function

```
PVStringPtr getDescriptor() const;
```

where

getDescriptor() Returns the `descriptor` field or null if the wrapped `PVStructure` has no `descriptor` field.

Example

```
NTScalarPtr scalar = NTScalar::createBuilder()->
    value(pvDouble)->
    addDescriptor()->create();

scalar->getDescriptor()->put("Beam current");
```

This produces:

```
epics:nt/NTScalar:1.0
    double value 0
    string descriptor Beam current
```

1.8.2 Normative Type support for alarm fields

Most Normative Types have an optional `alarm` field of the form

```
alarm_t alarm
    int severity
    int status
    string message
```

The corresponding Normative Type wrapper classes and their builders have support for this field:

Builder support

Each builder class for a Normative Type with an `alarm` field has a function

```
shared_pointer addAlarm();
```

where

addAlarm() Adds the `alarm` field to the structure returned by calling `create()`, `createStructure()` or `createPVStructure()`. Returns the the instance of the builder.

The effect of calling `addAlarm()` is reset by a call of `create()`, `createStructure()` or `createPVStructure()`.

Normative Type class support

Each wrapper class for a Normative Type with an `alarm` field has a function

```
bool attachAlarm(PVAlarm & pvAlarm) const;
PVStructurePtr getAlarm() const;
```

where

attachAlarm() Attaches the supplied `PVAlarm` to the wrapped `PVStructure`'s `alarm` field. Does nothing if no `alarm` field. Returns true if the operation was successful (i.e. the wrapped `PVStructure` has an `alarm` field), otherwise false.

getAlarm() Returns the `alarm` field or null if the wrapped `PVStructure` has no `alarm` field.

Example

```
NTScalarPtr scalar = NTScalar::createBuilder()->
    value(pvDouble)->
    addDescriptor()->create();

scalar->getValue<PVDouble>()->put(100.0);

PVAlarm pvAlarm;
scalar->attachAlarm(pvAlarm);

Alarm alarm;
alarm.setStatus(clientStatus);
alarm.setSeverity(majorAlarm);
alarm.setMessage("Too high");
pvAlarm.set(alarm);
```

This produces:

```
epics:nt/NTScalar:1.0
    double value 100
    alarm_t alarm
        int severity 2
        int status 7
        string message Too high
```

1.8.3 Normative Type support for timeStamp fields

Most Normative Types have an optional `timeStamp` field of the form

```
time_t timeStamp
    long secondsPastEpoch
    int nanoseconds
    int userTag
```

The corresponding Normative Type wrapper classes and their builders have support for this field:

Builder support

Each builder class for a Normative Type with a `timeStamp` field has a function

```
shared_pointer addTimeStamp();
```

where

addTimeStamp() Adds the `timeStamp` field to the structure returned by calling `create()`, `createStructure()` or `createPVStructure()`. Returns the the instance of the builder.

The effect of calling `addTimeStamp()` is reset by a call of `create()`, `createStructure()` or `createPVStructure()`.

Normative Type class support

Each wrapper class for a Normative Type with a `timeStamp` field has a function

```
bool attachTimeStamp(PVTimeStamp & pvTimeStamp) const;
PVStructurePtr getTimeStamp() const;
```

where

attachTimeStamp() Attaches a `PVTimeStamp` to the wrapped `PVStructure`'s `timeStamp` field. Does nothing if no `timeStamp` field. Returns true if the operation was successful (i.e. this instance has a `timeStamp` field), otherwise false.

getTimeStamp() Returns the `timeStamp` field or null if no `timeStamp` field.

Example

```
NTScalarPtr scalar = NTScalar::createBuilder()->
    value(pvDouble)->addTimeStamp()->create();
scalar->getValue<PVDouble>()->put(42);

PVTimeStamp pvTimeStamp;
scalar->attachTimeStamp(pvTimeStamp);

TimeStamp timeStamp;
timeStamp.getCurrent();
pvTimeStamp.set(timeStamp);
```

This will produce something like:

```
epics:nt/NTScalar:1.0
    double value 42
    time_t timeStamp
        long secondsPastEpoch 1473694453
        int nanoseconds 60324002
        int userTag 0
```

1.8.4 Normative Type support for display fields

Some Normative Types have an optional `display` field of the form

```
display_t display
    double limitLow
    double limitHigh
    string description
    string format
    string units
```

The corresponding Normative Type wrapper classes and their builders have support for this field:

Builder support

Each builder class for a Normative Type with a `display` field has a function

```
shared_pointer addDisplay();
```

where

addDisplay() Adds the `display` field to the structure returned by calling `create()`, `createStructure()` or `createPVStructure()`. Returns the the instance of the builder.

The effect of calling `addDisplay()` is reset by a call of `create()`, `createStructure()` or `createPVStructure()`.

Normative Type class support

Each wrapper class for a Normative Type with a `display` field has a function

```
bool attachDisplay(PVDisplay & pvDisplay) const;
PVStructurePtr getDisplay() const;
```

where

attachDisplay() Attaches a `PVDisplay` to the wrapped `PVDisplay`'s `display` field. Does nothing if no `display` field. Returns true if the operation was successful (i.e. this instance has a `display` field), otherwise false.

getDisplay() Returns the `display` field or null if no `display` field.

1.8.5 Normative Type support for control fields

Some Normative Types have an optional `control` field of the form

```
control_t control
    double limitLow
    double limitHigh
    double minStep
```

The corresponding Normative Type wrapper classes and their builders have support for this field:

Builder support

Each builder class for a Normative Type with a `control` field has a function

```
shared_pointer addControl();
```

where

addControl() Adds the `control` field to the structure returned by calling `create()`, `createStructure()` or `createPVStructure()`. Returns the the instance of the builder.

The effect of calling `addControl()` is reset by a call of `create()`, `createStructure()` or `createPVStructure()`.

Normative Type class support

Each wrapper class for a Normative Type with a `control` field has a function

```
bool attachControl(PVControl & pvControl) const;
PVStructurePtr getControl() const;
```

where

attachControl() Attaches a `PVControl` to the wrapped `PVControl`'s `control` field. Does nothing if no `control` field. Returns true if the operation was successful (i.e. this instance has a `control` field), otherwise false.

getControl() Returns the `control` field or null if no `control` field.

1.9 NTScalar

NTScalar is the EPICS 7 Normative Type that describes a single scalar value plus metadata:

Its structure is defined to be:

```
epics:nt/NTScalar:1.0
    scalar_t   value
    string descriptor                   : optional
    alarm_t alarm                       : optional
        int severity
        int status
        string message
    time_t timeStamp                    : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    display_t display                   : optional
        double limitLow
        double limitHigh
        string description
        string format
        string units
    control_t control                   : optional
        double limitLow
        double limitHigh
        double minStep
    {<field-type> <field-name>}0+  // additional fields
```

where scalar_t indicates a choice of scalar:

```
scalar_t :=

   boolean | byte |  ubyte |  short |  ushort |
   int |  uint |  long |  ulong |  float |  double |  string
```

1.9.1 NTScalarBuilder

This is a class that creates the introspection and data instances for NTScalar and an a NTScalar instance itself.

ntscalar.h defines the following:

```
class NTScalar;
typedef std::tr1::shared_ptr<NTScalar> NTScalarPtr;

class NTScalarBuilder
{
public:
    POINTER_DEFINITIONS(NTScalarBuilder);
    shared_pointer value(ScalarType scalarType);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    shared_pointer addDisplay();
    shared_pointer addControl();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTScalarPtr create();
    shared_pointer add(
         string const & name,
         FieldConstPtr const & field);
private:
    // ... remainder of class definition
}
```

where

value Sets the scalar type for the `value` field. This must be specified or a call of `create()`, `createStructure()` or `createPVStructure()` will throw an exception (`std::runtime_error`).

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTScalarArrayBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTScalar` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the effect of calling `value()` as well all calls of optional field/property data functions and additional field functions.

NTScalarBuilder Examples

An example of creating an NTScalar instance is:

```
NTScalarBuilderPtr builder = NTScalar::createBuilder();
NTScalarPtr ntScalar = builder->
    value(pvInt)->
    addDescriptor()->
    addAlarm()->
    addTimeStamp()->
    addDisplay()->
    addControl()->
    create();
```

1.9.2 NTScalar

ntscalar.h defines the following:

```
class NTScalar;
typedef std::tr1::shared_ptr<NTScalar> NTScalarPtr;

class NTScalar
{
public:
    POINTER_DEFINITIONS(NTScalar);
    ~NTScalar() {}
    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTScalarBuilderPtr createBuilder();

    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachAlarm(PVAlarm &pvAlarm) const;
    bool attachDisplay(PVDisplay &pvDisplay) const;
    bool attachControl(PVControl &pvControl) const;

    PVStructurePtr getPVStructure() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;
    PVStructurePtr getDisplay() const;
    PVStructurePtr getControl() const;

    PVFieldPtr getValue() const;

    template<typename PVT>
    std::tr1::shared_ptr<PVT> getValue() const
private:
    // ... remainder of class definition
}
```

where

getValue() Returns the `value` field. The template version returns the type supplied in the template argument.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.10 NTScalarArray

NTScalarArray is the EPICS 7 Normative Type that describes an array of values, plus metadata. All the elements of the array of the same scalar type.

```
epics:nt/NTScalarArray:1.0
    scalar_t[]   value
    string descriptor              : optional
    alarm_t alarm                  : optional
        int severity
        int status
        string message
    time_t timeStamp               : optional
```

(continues on next page)

(continued from previous page)

```
        long secondsPastEpoch
        int nanoseconds
        int userTag
    display_t display              : optional
        double limitLow
        double limitHigh
        string description
        string format
        string units
    {<field-type> <field-name>}0+  // additional fields
```

where scalar_t[] indicates a choice of scalar array:

```
scalar_t[] :=

   boolean[] | byte[] |  ubyte[] |  short[] |  ushort[] |
   int[] |  uint[] |  long[] |  ulong[] |  float[] |  double[] |  string[]
```

1.10.1 NTScalarArrayBuilder

ntscalarArray.h defines the following:

```
class NTScalarArray;
typedef std::tr1::shared_ptr<NTScalarArray> NTScalarArrayPtr;

class NTScalarArrayBuilder
{
public:
    POINTER_DEFINITIONS(NTScalarArrayBuilder);
    shared_pointer value(ScalarType elementType);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    shared_pointer addDisplay();
    shared_pointer addControl();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTScalarArrayPtr create();
    shared_pointer add(
         string const & name,
         FieldConstPtr const & field);
private:
    // ... remainder of class definition
};
```

where

value Sets the element type for the `value` field. This must be specified or a call of `create()`, `createStructure()` or `createPVStructure()` will throw an exception (`std::runtime_error`).

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTScalarArrayBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTScalarArray` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the effect of calling `value()` as well all calls of optional field/property data functions and additional field functions.

1.10.2 NTScalarArray

ntscalarArray.h defines the following:

```
class NTScalarArray;
typedef std::tr1::shared_ptr<NTScalarArray> NTScalarArrayPtr;

class NTScalarArray
{
public:
    POINTER_DEFINITIONS(NTScalarArray);
    ~NTScalarArray() {}
    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTScalarArrayBuilderPtr createBuilder();

    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachAlarm(PVAlarm &pvAlarm) const;
    bool attachDisplay(PVDisplay &pvDisplay) const;
    bool attachControl(PVControl &pvControl) const;

    PVStructurePtr getPVStructure() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;
    PVStructurePtr getDisplay() const;
    PVStructurePtr getControl() const;

    PVFieldPtr getValue() const;
    template<typename PVT>
    std::tr1::shared_ptr<PV> getValue() const
private:
    // ... remainder of class definition
};
```

where

getValue Returns the `value` field. The template version returns the type supplied in the template argument.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.11 NTEnum

NTEnum is an EPICS 7 Normative Type that describes an enumeration (a closed set of possible values specified by an n-tuple).

Its structure is defined to be:

```
epics:nt/NTEnum:1.0
    enum_t value
        int index
        string[] choices
    string descriptor                : optional
    alarm_t alarm                    : optional
        int severity
        int status
        string message
    time_t timeStamp                 : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    {<field-type> <field-name>}0+ // additional fields
```

1.11.1 NTEnumBuilder

ntscalarArray.h defines the following:

```
class NTEnum;
typedef std::tr1::shared_ptr<NTEnum> NTEnumPtr;

class NTEnumBuilder
{
public:
    POINTER_DEFINITIONS(NTEnumBuilder);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTEnumPtr create();
    shared_pointer add(string const & name, FieldConstPtr const & field);

private:
     // ... remainder of class definition
};
```

where all functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTEnumBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTEnum` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes all calls of optional field/property data functions and additional field functions.

1.11.2 NTEnum

ntenum.h defines the following:

```
class NTEnum
{
public:
    POINTER_DEFINITIONS(NTEnum);
    static const string URI;
```

(continues on next page)

(continued from previous page)

```
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTEnumBuilderPtr createBuilder();
    getPVStructure() const;

    attachTimeStamp(PVTimeStamp & pvTimeStamp) const;
    attachAlarm(PVAlarm & pvAlarm) const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;

    PVStructurePtr getValue() const;

private:
    // ... remainder of class definition
};
```

where

getValue Returns the `value` field.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.12 NTMatrix

NTMatrix is an EPICS 7 Normative Type used to define a matrix, specifically a 2-dimensional array of real numbers.

Its structure is defined to be:

```
epics:nt/NTMatrix:1.0
    double[]    value
    int[2]      dim                 :optional
    string      descriptor          :optional
    alarm_t alarm                   :optional
        int severity
        int status
        string message
    time_t timeStamp                : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    display_t display               : optional
        double limitLow
        double limitHigh
        string description
        string format
        string units
    {<field-type> <field-name>}0+  // additional fields
```

1.12.1 NTMatrixBuilder

ntmatrix.h defines the following:

```
class NTMatrixBuilder
{
public:
    POINTER_DEFINITIONS(NTMatrixBuilder);

    shared_pointer addDim();

    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    shared_pointer addDisplay();

    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    shared_pointer add(string const & name, FieldConstPtr const & field);

private:
    // ... remainder of class definition
};
```

where

addDim Adds optional `dimension` field.

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTMatrixBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTMatrix` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the effect of calling `addDim()` as well all calls of optional field/property data functions and additional field functions.

1.12.2 NTMatrix

ntmatrix.h defines the following:

```
class NTMatrix
{
public:
    POINTER_DEFINITIONS(NTMatrix);

    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTMatrixBuilderPtr createBuilder();

    bool attachTimeStamp(PVTimeStamp & pvTimeStamp) const;
    bool attachAlarm(PVAlarm & pvAlarm) const;
    bool attachDisplay(PVDisplay & pvDisplay) const;
```

(continues on next page)

(continued from previous page)

```
    PVStructurePtr getPVStructure() const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;
    PVStructurePtr getDisplay() const;

    PVDoubleArrayPtr getValue() const;
    PVIntArrayPtr getDim() const;

private:
    // ... remainder of class definition
};
```

where

getValue Returns the `value` field.

getDim Returns the `dim` field.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.13 NTURI

NTURI is the EPICS 7 Normative Type that describes a Uniform Resource Identifier (URI).

Its structure is defined to be:

```
epics:nt/NTURI:1.0
    string scheme
    string authority               : optional
    string path
    structure query                : optional
        {string | double | int <field-name>}0+
    {<field-type> <field-name>}0+  // additional fields
```

1.13.1 NTURIBuilder

nturi.h defines the following:

```
class NTURI;
typedef std::tr1::shared_ptr<NTURI> NTURIPtr;

classNTURIBuilder
{
public:
    POINTER_DEFINITIONS(NTURIBuilder);

    shared_pointer addAuthority();
    shared_pointer addQueryString(string const & name);
    shared_pointer addQueryDouble(string const & name);
    shared_pointer addQueryInt(string const & name);

    StructureConstPtr createStructure();
```

(continues on next page)

(continued from previous page)

```
    PVStructurePtr createPVStructure();
    NTURIPtr create();
    shared_pointer add(string const & name, FieldConstPtr const & field);

private:
    // ... remainder of class definition
};
```

where

addAuthority Adds optional `dimension` field.

addQueryString Adds a string field of the supplied name to the optional `query` field.

addQueryDouble Adds a double field of the supplied name to the optional `query` field.

addQueryDouble Adds an integer field of the supplied name to the optional `query` field.

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTURIBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTURI` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the effect of calling `addAuthority()` and the 3 "add query" functions.

1.13.2 NTURI

nturi.h defines the following:

```
class NTURI
{
public:
    POINTER_DEFINITIONS(NTURI);

    static const string URI;

    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTURIBuilderPtr createBuilder();

    PVStructurePtr getPVStructure() const;
    PVStringPtr getScheme() const;
    PVStringPtr getAuthority() const;
    PVStringPtr getPath() const;
    PVStructurePtr getQuery() const;

    StringArray const & getQueryNames() const;
    PVFieldPtr getQueryField(string const & name) const;
    template<typename PVT>
    std::tr1::shared_ptr<PVT> getQueryField(string const & name) const;

private:
```

(continues on next page)

(continued from previous page)

```
    // ... remainder of class definition
};
```

where

getScheme() Returns the `scheme` field.

getAuthority() Returns the optional `authority` field.

getPath() Returns the `path` field.

getQuery() Returns the optional `query` field.

getQueryNames() Returns the names of the fields of the `query` field.

getQueryField() Returns the subfield of the `query` field with the requested name. The template version returns the type requested in the template argument.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.14 NTNameValue

NTNameValue is the EPICS 7 Normative Type that describes a system of name and scalar values.

Its structure is defined to be:

```
epics:nt/NTNameValue:1.0
    string[] name
    double[] value
    string descriptor              : optional
    alarm_t alarm                  : optional
        int severity
        int status
        string message
    time_t timeStamp               : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    {<field-type> <field-name>}0+  // additional fields
```

1.14.1 NTNameValueBuilder

ntnameValue.h defines the following:

```
class NTNameValue;
typedef std::tr1::shared_ptr<NTNameValue> NTNameValuePtr;

class NTNameValueBuilder
{
public:
    POINTER_DEFINITIONS(NTNameValueBuilder);
    shared_pointer value(ScalarType scalarType);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
```

(continues on next page)

(continued from previous page)

```
    shared_pointer addTimeStamp();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTNameValuePtr create();
    shared_pointer add(
         string const & name,
         FieldConstPtr const & field);
private:
    // ... remainder of class definition
};
```

where

value Sets the scalar type for the `value` field. This must be specified or a call of `create()`, `createStructure()` or `createPVStructure()` will throw an exception (`std::runtime_error`)

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTNameValueBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTNameValue` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the effect of calling `value()` as well all calls of optional field/property data functions and additional field functions.

1.14.2 NTNameValue

ntnameValue.h defines the following:

```
class NTNameValue;
typedef std::tr1::shared_ptr<NTNameValue> NTNameValuePtr;

class NTNameValue
{
public:
    POINTER_DEFINITIONS(NTNameValue);
    ~NTNameValue() {}
    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTNameValueBuilderPtr createBuilder();

    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachAlarm(PVAlarm &pvAlarm) const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getPVStructure() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;
    PVStringArrayPtr getName() const;
    PVFieldPtr getValue() const;
    template<typename PVT>
    std::tr1::shared_ptr<PV> getValue() const
```

(continues on next page)

(continued from previous page)

```
private:
    // ... remainder of class definition
}
```

where

getName Returns the `name` field.

getValue Returns the `value` field.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.15 NTTable

NTTable is the EPICS 7 Normative Type suitable for column-oriented tabular datasets.

Its structure is defined to be:

```
epics:nt/NTTable:1.0
    string[] labels []
    structure value
        {column_t[]  colname}0+ // 0 or more scalar array instances, the column␣
↪values.
    string descriptor              : optional
    alarm_t alarm                  : optional
        int severity
        int status
        string
    time_t timeStamp               : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    {<field-type> <field-name>}0+  // additional fields
```

1.15.1 NTTableBuilder

nttable.h defines the following:

```
class NTTable;
typedef std::tr1::shared_ptr<NTTable> NTTablePtr;

class NTTableBuilder
{
public:
    POINTER_DEFINITIONS(NTTableBuilder);
    shared_pointer addColumn(string const & name,ScalarType scalarType);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTTablePtr create();
    shared_pointer add(
```

(continues on next page)

(continued from previous page)

```
        string const & name,
        FieldConstPtr const & field);
private:
    // ... remainder of class definition
}
```

where

addColumn Adds a column (subfield of `value` field) of the specified name and scalar type

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTTableBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTTable` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the added columns as well all calls of optional field/property data functions and additional field functions.

1.15.2 NTTable

nttable.h defines the following:

```
class NTTable;
typedef std::tr1::shared_ptr<NTTable> NTTablePtr;

class NTTable
{
public:
    POINTER_DEFINITIONS(NTTable);
    ~NTTable() {}
    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTTableBuilderPtr createBuilder();

    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachAlarm(PVAlarm &pvAlarm) const;
    PVStructurePtr getPVStructure() const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;
    PVStringArrayPtr getLabels() const;
    PVFieldPtr getColumn(string const & columnName) const;
    template<typename PVT>
    std::tr1::shared_ptr<PV> getColumn(string const & columnName) const;
private:
    // ... remainder of class definition
}
```

where

getLabels Returns the labels field.

getColumn Returns the column with the specified name.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.16 NTAttribute

NTAttribute is the EPICS 7 Normative Type for a named attribute of any type. It is is essentially a key-value pair which optionally can be tagged with additional strings.

Its structure is defined to be:

```
epics:nt/NTAttribute:1.0
    string    name
    any       value
    string[]  tags                : optional
    string    descriptor          : optional
    alarm_t alarm                 : optional
        int severity
        int status
        string
    time_t timeStamp              : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    {<field-type> <field-name>}0+  // additional fields
```

1.16.1 NTAttributeBuilder

ntattribute.h defines the following:

```
class NTAttribute;
typedef std::tr1::shared_ptr<NTAttribute> NTAttributePtr;

class  NTAttributeBuilder
{
public:
    POINTER_DEFINITIONS(NTAttributeBuilder);

    shared_pointer addTags();
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();

    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTAttributePtr create();
    shared_pointer add(string const & name, FieldConstPtr const & field);

    protected:
    // ... remainder of class definition
}
```

where

addTags Adds optional tags field.

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTAttribute` can be used to create multiple `Structure`, `PVStructure` and/or `NTAttribute` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes any call of `addTags()` as well as calls of optional field/property data functions and additional field functions.

1.16.2 NTAttribute

ntattribute.h defines the following:

```
class NTAttribute;
typedef std::tr1::shared_ptr<NTAttribute> NTAttributePtr;

class NTAttribute
{
public:
    POINTER_DEFINITIONS(NTAttribute);

    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(
        StructureConstPtr const & structure);
    static bool isCompatible(
        PVStructurePtr const & pvStructure);
    bool isValid();
    static NTAttributeBuilderPtr createBuilder();
    ~NTAttribute() {}
    bool attachTimeStamp(PVTimeStamp & pvTimeStamp) const;
    bool attachAlarm(PVAlarm & pvAlarm) const;
    PVStructurePtr getPVStructure() const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;

    PVStringPtr getName() const;
    PVUnionPtr getValue() const;
    PVStringArrayPtr getTags() const;

private:
    // ... remainder of class definition
}
```

where

getName() Returns the `labels` field.

getValue() Returns the `value` field.

getTags() Returns the optional `tags` field.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.17 NTAttribute extended for NDArray

Support is provided for the NTAttribute Normative Type extended as required by NTNDArray.

The structure of is defined to be:

```
epics:nt/NTAttribute:1.0
    string    name
    any       value
    string[]  tags                : optional
    string    descriptor
    alarm_t alarm                 : optional
        int severity
        int status
        string
    time_t timeStamp              : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    int       sourceType
    string    source
    {<field-type> <field-name>}0+  // additional fields
```

This is as NTAttribute except the standard additional fields `sourceType` and `source` have been added and `descriptor` is no longer optional.

The builder and wrapper classes are `NTNDArrayAttributeBuilder` and `NTNDArrayAttribute` respectively.

These are defined in **ntndarrayAttribute.h**.

The class definitions are the same except that:

1. `NTNDArrayAttribute::addDescriptor()` is a null-op, as `descriptor` is no longer optional
2. `isCompatible()` checks that the the structure is conformant with respect to the extension required by NTNDArray (i.e. it has conformant `descriptor`, `sourceType` and `source` fields) and
3. two new functions are provided for accessing the `sourceType` and `source` fields:

```
class NTNDArrayAttribute
{
public:
    // ...
    PVIntPtr getSourceType() const;
    getSource() const;
    // ...
};
```

1.18 NTMultiChannel

NTMultiChannel is an EPICS 7 Normative Type that aggregates an array of values from different EPICS Process Variable (PV) channel sources, not necessarily of the same type, into a single variable.

Its structure is defined to be:

```
epics:nt/NTMultiChannel:1.0
    anyunion_t[] value
    string[] channelName
    alarm_t alarm                   : optional
        int severity
        int status
        string
    time_t timeStamp                : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    int[] severity                  : optional
    int[] status                    : optional
    string[] message                : optional
    long[] secondsPastEpoch         : optional
    int[] nanoseconds               : optional
    string descriptor               : optional
    {<field-type> <field-name>}0+  // additional fields
```

where anyunion_t[] means any union array - either a variant union array or any choice of regular union array.

1.18.1 NTMultiChannelBuilder

ntmultiChannel.h defines the following:

```
class NTMultiChannel;
typedef std::tr1::shared_ptr<NTMultiChannel> NTMultiChannelPtr;

class NTMultiChannelBuilder
{
public:
    POINTER_DEFINITIONS(NTMultiChannelBuilder);
    shared_pointer value(UnionConstPtr valuePtr);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    shared_pointer addSeverity();
    shared_pointer addStatus();
    shared_pointer addMessage();
    shared_pointer addSecondsPastEpoch();
    shared_pointer addNanoseconds();
    shared_pointer addUserTag();
    shared_pointer addIsConnected();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTMultiChannelPtr create();
    shared_pointer add(
         string const & name,
         FieldConstPtr const & field);
private:
}
```

where

value Sets the element type for the `value` field. If not specified the type will be a variant union.

addSeverity() Add a field that has the alarm severity for each channel.

addStatus() Add a field that has the alarm status for each channel.

addMessage() Add a field that has the alarm message for each channel.

addSecondsPastEpoch() Add a field that has the secondsPastEpoch for each channel.

addNanoseconds() Add a field that has the nanoseconds for each channel.

addUserTag() Add a field that has the userTag for each channel.

addIsConnected() Add a field that has the connection state for each channel. (Not an optional field of the type, but commonly included.)

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTMultiChannelBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTMultiChannel` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the union specified by `value()` (which is reset to a variant union) and all calls to add NTMultiChannel optional fields (including all optional field/property data functions) and additional fields.

1.18.2 NTMultiChannel

ntmultiChannel.h defines the following:

```
class NTMultiChannel;
typedef std::tr1::shared_ptr<NTMultiChannel> NTMultiChannelPtr;

class NTMultiChannel
{
public:
    POINTER_DEFINITIONS(NTMultiChannel);
    ~NTMultiChannel() {}
    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTMultiChannelBuilderPtr createBuilder();

    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachAlarm(PVAlarm &pvAlarm) const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getPVStructure() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;
    PVUnionArrayPtr getValue() const;
    PVStringArrayPtr getChannelName() const;
    PVBooleanArrayPtr getIsConnected() const;
    PVIntArrayPtr getSeverity() const;
    PVIntArrayPtr getStatus() const;
    PVStringArrayPtr getMessage() const;
    PVLongArrayPtr getSecondsPastEpoch() const;
    PVIntArrayPtr getNanoseconds() const;
    PVIntArrayPtr getUserTag() const;
```

(continues on next page)

(continued from previous page)

```
private:
}
```

where

getValue() Returns the `value` field.

getChannelName() Returns the `name` field. (Contains the name of each channel.)

getIsConnected() Returns the additional `isConnected` field. (Contains the connection state of each channel.) This is not an optional field of the type, but is commonly included.

getSeverity() Returns the `severity` field. (Contains the alarm severity of each channel.)

getStatus() Returns the `status` field. (Contains the alarm status of each channel.)

getMessage() Returns the `message` field. (Contains the alarm message of each channel.)

getSecondsPastEpoch() Returns the `secondsPastEpoch` field. (Contains the timeStamp secondsPastEpoch of each channel.)

getNanoseconds() Returns the `nanoseconds` field. (Contains the timeStamp nanoseconds of each channel.)

getUserTag() Returns the `userTag` field. (Contains the timeStamp userTag of each channel.)

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.19 NTNDArray

NTNDArray is an EPICS Version 4 Normative Type designed to encode data from detectors and cameras, especially areaDetector applications. The type is heavily modeled on areaDetector's NDArray class. One NTNDArray gives one frame.

Its structure is defined to be:

```
epics:nt/NTNDArray:1.0
    union value
        boolean[] booleanValue
        byte[]    byteValue
        short[]   shortValue
        int[]     intValue
        long[]    longValue
        ubyte[]   ubyteValue
        ushort[]  ushortValue
        uint[]    uintValue
        ulong[]   ulongValue
        float[]   floatValue
        double[]  doubleValue
    codec_t codec
        string name
        any parameters
    long compressedSize
    long uncompressedSize
    dimension_t[] dimension
        dimension_t[]
            dimension_t
                int size
```

(continues on next page)

(continued from previous page)

```
                int offset
                int fullSize
                int binning
                boolean reverse
    int uniqueId
    time_t dataTimeStamp
        long secondsPastEpoch
        int nanoseconds
        int userTag
    epics:nt/NTAttribute:1.0[] attribute
        epics:nt/NTAttribute:1.0[]
            epics:nt/NTAttribute:1.0
                string name
                any value
                string description
                int sourceType
                string source
    string descriptor               : optional
    time_t timeStamp                : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    alarm_t alarm                   : optional
        int severity
        int status
        string message
    display_t display               : optional
        double limitLow
        double limitHigh
        string description
        string format
        string units
    {<field-type> <field-name>}0+  // additional fields
```

1.19.1 NTNDArrayBuilder

ntndArray.h defines the following:

```
class NTNDArray;
typedef std::tr1::shared_ptr<NTNDArray> NTNDArrayPtr;

class NTNDArrayBuilder
{
public:
    POINTER_DEFINITIONS(NTNDArrayBuilder);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    shared_pointer addDisplay();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTNDArrayPtr create();
    shared_pointer add(
         string const & name,
         FieldConstPtr const & field);
```

(continues on next page)

(continued from previous page)

```
private:
    // ... remainder of class definition
}
```

where all functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

1.19.2 NTNDArray

```
class NTNDArray;
typedef std::tr1::shared_ptr<NTNDArray> NTNDArrayPtr;

class NTNDArray
{
public:
    POINTER_DEFINITIONS(NTNDArray);
    ~NTNDArray() {}
    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTNDArrayBuilderPtr createBuilder();

    PVStringPtr getDescriptor() const;
    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachDataTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachAlarm(PVAlarm &pvAlarm) const;
    PVStructurePtr getPVStructure() const;
    PVUnionPtr getValue() const;
    PVStructurePtr getCodec() const;
    PVLongPtr getCompressedDataSize() const;
    PVLongPtr getUncompressedDataSize() const;
    PVStructureArrayPtr getAttribute() const;
    PVStructureArrayPtr getDimension() const;
    PVIntPtr getUniqueId() const;
    PVStructurePtr getDataTimeStamp() const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;
    PVStructurePtr getDisplay() const;
private:
    // ... remainder of class definition
}
```

where

attachDataTimeStamp() Attaches a `PVTimeStamp` to the wrapped `PVStructure`'s `timeStamp` field. Does nothing if no `timeStamp` field. Returns true if the operation was successful (i.e. this instance has a `timeStamp` field), otherwise false.

getValue() Returns the `value` field.

getCodec Returns `codec` field.

getCompressedDataSize Returns `compressedDataSize` field.

getUncompressedDataSize Returns `uncompressedDataSize` field.

getAttribute Returns the `attribute` field.

getDimension Returns the `dimension` field.

getUniqueId Returns the `uniqueId` field.

getDataTimeStamp Returns the `dataTimeStamp`.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.20 NTContinuum

NTContinuum is the EPICS 7 Normative Type used to express a sequence of point values in time or frequency domain. Each point has N values (N>=1) and an additional value which describes the index of the list. The additional value is carried in the base field. The value field carries the values which make up the point in index order.

Its structure is defined to be:

```
epics:nt/NTContinuum:1.0
    double[]   base
    double[]   value
    string[]   units
    string descriptor               : optional
    alarm_t alarm                   : optional
        int severity
        int status
        string message
    time_t timeStamp                : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
```

1.20.1 NTContinuumBuilder

ntcontinuum.h defines the following:

```
class NTContinuum;
typedef std::tr1::shared_ptr<NTContinuum> NTContinuumPtr;


class NTContinuumBuilder
{
public:
    POINTER_DEFINITIONS(NTContinuumBuilder);

    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTContinuumPtr create();
```

(continues on next page)

(continued from previous page)

```
    shared_pointer add(std::string const & name, FieldConstPtr const & field);

    private:
    // ... remainder of class definition
};
```

where all functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTContinuumBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTContinuum` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes all calls to add optional fields (including property fields) and additional fields.

1.20.2 NTContinuum

ntcontinuum.h defines the following:

```
class NTContinuum
{
public:
    POINTER_DEFINITIONS(NTContinuum);

    static const std::string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(
        StructureConstPtr const &structure);
    static bool isCompatible(
        PVStructurePtr const &pvStructure);
    bool isValid();
    static NTContinuumBuilderPtr createBuilder();
    ~NTContinuum() {}

    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachAlarm(PVAlarm &pvAlarm) const;

    PVStructurePtr getPVStructure() const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;
    PVDoubleArrayPtr getBase() const;
    PVDoubleArrayPtr getValue() const;
    PVStringArrayPtr getUnits() const;

private:
    // ... remainder of class definition
};
```

where

getBase() Returns the `base` field.

getValue() Returns the `value` field.

getUnits() Returns the `units()` field.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.21 NTHistogram

NTHistogram is the EPICS 7 Normative Type used to encode the data and representation of a (1-dimensional) histogram. Specifically, it encapsulates frequency binned data.

Its structure is defined to be:

```
epics:nt/NTHistogram:1.0
   double[] ranges
    (short[] | int[] | long[]) value
    string descriptor                : optional
    alarm_t alarm                    : optional
        int severity
        int status
        string message
    time_t timeStamp                 : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
```

1.21.1 NTHistogramBuilder

nthistogram.h defines the following:

```
class NTHistogramBuilder
{
public:
    POINTER_DEFINITIONS(NTHistogramBuilder);

    shared_pointer value(ScalarType scalarType);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTHistogramPtr create();
    shared_pointer add(std::string const  &  name, FieldConstPtr const  &  field);

private:
    // ... remainder of class definition
};
```

where

value This sets the element type for the `value` field (short, int or long). This must be specified or a call of `create()`, `createStructure()` or `createPVStructure()` will throw an exception (`std::runtime_error`).

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTHistogramBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTHistogram` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the scalar type specified by `value()` and all calls to add optional field/property data functions and additional fields.

1.21.2 NTHistogram

nthistogram.h defines the following:

```
class NTHistogram
{
public:
    POINTER_DEFINITIONS(NTHistogram);

    static const std::string URI;
    static shared_pointer wrap(PVStructurePtr const  &  pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const  &  pvStructure);
    static bool is_a(StructureConstPtr const  &  structure);
    static bool is_a(PVStructurePtr const  &  pvStructure);
    static bool isCompatible(StructureConstPtr const  & structure);
    static bool isCompatible(PVStructurePtr const  & pvStructure);
    bool isValid();
    static NTHistogramBuilderPtr createBuilder();
    ~NTHistogram() {}

    bool attachTimeStamp(PVTimeStamp  & pvTimeStamp) const;
    bool attachAlarm(PVAlarm  & pvAlarm) const;

    PVStructurePtr getPVStructure() const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;

    PVDoubleArrayPtr getRanges() const;

    PVScalarArrayPtr getValue() const;

    template<typename PVT>
    std::tr1::shared_ptr<PVT> getValue() const;

private:
    // ... remainder of class definition
};
```

where

getRanges() Returns the `ranges` field.

getValue() Returns the `value` field. The template version returns the type supplied in the template argument.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.22 NTAggregate

NTAggregate is the EPICS 7 Normative Type to compactly convey data which combines several measurements or observation. NTAggregate gives simple summary statistic about the central tendency and dispersion of a set of data points.

Its structure is defined to be:

```
epics:nt/NTAggregate:1.0
    double value
    long N
    double dispersion              : optional
    double first                   : optional
    time_t firstTimeStamp          : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    time_t lastTimeStamp           : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    double max                     :optional
    double min                     :optional
    string descriptor                : optional
    alarm_t alarm                    : optional
        int severity
        int status
        string message
    time_t timeStamp                 : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    {<field-type> <field-name>}0+  // additional fields
```

1.22.1 NTAggregateBuilder

ntaggregate.h defines the following:

```
class NTAggregate;
typedef std::tr1::shared_ptr<NTAggregate> NTAggregatePtr;

class NTAggregateBuilder
{
public:
    POINTER_DEFINITIONS(NTAggregateBuilder);
    shared_pointer addDispersion();
    shared_pointer addFirst();
    shared_pointer addFirstTimeStamp();
    shared_pointer addLast();
    shared_pointer addLastTimeStamp();
    shared_pointer addMax();
    shared_pointer addMin();

    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
```

(continues on next page)

(continued from previous page)

```
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTAggregatePtr create();

    shared_pointer add(std::string const & name, FieldConstPtr const & field);

private:
    // ... remainder of class definition
};
```

where

addDispersion() Adds optional `dispersion` field.

addFirst() Adds optional `first` field.

addFirstTimeStamp() Adds optional `firstTimeStamp` field.

addLast() Adds optional `last` field.

addLastTimeStamp() Adds optional `lastTimeStamp` field.

addMax() Adds optional `max` field.

addMin() Adds optional `min` field.

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTAggregateBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTAggregate` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes all calls to add optional fields (including property fields) and additional fields.

1.22.2 NTAggregate

ntaggregate.h defines the following:

```
class NTAggregate
{
public:
    POINTER_DEFINITIONS(NTAggregate);

    static const std::string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(
        StructureConstPtr const &structure);
    static bool isCompatible(
        PVStructurePtr const &pvStructure);
    bool isValid();
    static NTAggregateBuilderPtr createBuilder();

    ~NTAggregate() {}

    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
```

(continues on next page)

(continued from previous page)

```
    bool attachAlarm(PVAlarm &pvAlarm) const;

    PVStructurePtr getPVStructure() const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;

    PVDoublePtr getValue() const;
    PVLongPtr getN() const;
    PVDoublePtr getDispersion() const;
    PVDoublePtr getFirst() const;
    PVStructurePtr getFirstTimeStamp() const;
    PVDoublePtr getLast() const;
    PVStructurePtr getLastTimeStamp() const
    PVDoublePtr getMax() const;
    PVDoublePtr getMin() const;

private:
    // ... remainder of class definition
};
```

where

getValue() Returns the `value` field.

getN() Returns the `N` field.

getDispersion() <to do>

getFirst() Returns the `first` field.

getFirstTimeStamp() Returns the `firstTimeStamp` field.

getLast() Returns the `last` field.

getLastTimeStamp() Returns the `lastTimeStamp` field.

getMax() Returns the `max` field.

getMin() Returns the `min` field.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.23 NTUnion

NTUnion is a Normative Type for interoperation of essentially any data structure, plus description, alarm and timestamp fields.

Its structure is defined to be:

```
epics:nt/NTUnion:1.0
    anyunion_t value
    string descriptor              : optional
    alarm_t alarm                  : optional
        int severity
        int status
        string message
```

(continues on next page)

(continued from previous page)

```
    time_t timeStamp                : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    {<field-type> <field-name>}0+  // additional fields
```

1.23.1 NTUnionBuilder

ntunion.h defines the following:

```
class NTUnion;
typedef std::tr1::shared_ptr<NTUnion> NTUnionPtr;

class NTUnionBuilder
{
public:
    POINTER_DEFINITIONS(NTUnionBuilder);
    shared_pointer value(UnionConstPtr valuePtr);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTUnionPtr create();
    shared_pointer add( string const & name, FieldConstPtr const & field);

private:
    // ... remainder of class definition
}
```

where

value This determines the element type for the `value` field. If not specified the type will be a variant union.

An `NTUnionBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTUnion` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the union specified by `value()` (which is reset to a variant union) and all calls to add optional field/property data functions and additional fields.

1.23.2 NTUnion

ntunion.h defines the following:

```
class NTUnion;
typedef std::tr1::shared_ptr<NTUnion> NTUnionPtr;

class NTUnion
{
public:
    POINTER_DEFINITIONS(NTUnion);
    ~NTUnion() {}
    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
```

(continues on next page)

(continued from previous page)

```
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    bool isValid();
    static NTUnionBuilderPtr createBuilder();
    getPVStructure() const;

    attachTimeStamp(PVTimeStamp & pvTimeStamp) const;
    attachAlarm(PVAlarm & pvAlarm) const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;

    PVUnionPtr getValue() const;
private:
    // ... remainder of class definition
}
```

where

getValue Returns the `value` field.

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.

1.24 NTScalarMultiChannel

NTScalarMultiChannel is an EPICS 7 Normative Type that aggregates an array of values from different EPICS Process Variable (PV) channel sources of the same scalar type into a single variable.

Its structure is defined to be:

```
epics:nt/NTScalarMultiChannel:1.0
    scalar_t[] value
    string[] channelName
    alarm_t alarm                   : optional
        int severity
        int status
        string
    time_t timeStamp                : optional
        long secondsPastEpoch
        int nanoseconds
        int userTag
    int[] severity                  : optional
    int[] status                    : optional
    string[] message                : optional
    long[] secondsPastEpoch         : optional
    int[] nanoseconds               : optional
    string descriptor               : optional
    {<field-type> <field-name>}0+   // additional fields
```

where scalar_t[] indicates a choice of scalar array:

```
scalar_t[] :=

   boolean[] | byte[] |  ubyte[] |  short[] |  ushort[] |
   int[] |  uint[] |  long[] |  ulong[] |  float[] |  double[] |  string[]
```

1.24.1 NTScalarMultiChannelBuilder

ntscalarMultiChannel.h defines the following:

```
class NTScalarMultiChannel;
typedef std::tr1::shared_ptr<NTScalarMultiChannel> NTScalarMultiChannelPtr;

class NTScalarMultiChannelBuilder
{
public:
    POINTER_DEFINITIONS(NTScalarMultiChannelBuilder);
    shared_pointer value(ScalarType scalarType);
    shared_pointer addDescriptor();
    shared_pointer addAlarm();
    shared_pointer addTimeStamp();
    shared_pointer addSeverity();
    shared_pointer addStatus();
    shared_pointer addMessage();
    shared_pointer addSecondsPastEpoch();
    shared_pointer addNanoseconds();
    shared_pointer addUserTag();
    StructureConstPtr createStructure();
    PVStructurePtr createPVStructure();
    NTScalarMultiChannelPtr create();
    shared_pointer add(
         string const & name,
         FieldConstPtr const & field);
private:
}
```

where

value This determines the element type for the `value` field. If not specified the type will double.

addSeverity() Add a field that has the alarm severity for each channel.

addStatus() Add a field that has the alarm status for each channel.

addMessage() Add a field that has the alarm message for each channel.

addSecondsPastEpoch() Add a field that has the secondsPastEpoch for each channel.

addNanoseconds() Add a field that has the nanoseconds for each channel.

addUserTag() Add a field that has the userTag for each channel.

addIsConnected() Add a field that has the connection state for each channel. (Not an optional field of the type, but commonly included.)

and all other functions are described in the sections *Features common to all Normative Type Builder classes* and *Normative Type Property Features*.

An `NTScalarMultiChannelBuilder` can be used to create multiple `Structure`, `PVStructure` and/or `NTScalarMultiChannel` instances.

A call of `create()`, `createStructure()` or `createPVStructure()` clears all internal data. This includes the scalar type specified by `value()` (which is reset to double) and all calls to add NTScalarMultiChannel optional fields (including all optional field/property data functions) and additional fields.

1.24.2 NTScalarMultiChannel

ntscalarMultiChannel.h defines the following:

```
class NTScalarMultiChannel;
typedef std::tr1::shared_ptr<NTScalarMultiChannel> NTScalarMultiChannelPtr;

class NTScalarMultiChannel
{
public:
    POINTER_DEFINITIONS(NTScalarMultiChannel);
    ~NTScalarMultiChannel() {}
    static const string URI;
    static shared_pointer wrap(PVStructurePtr const & pvStructure);
    static shared_pointer wrapUnsafe(PVStructurePtr const & pvStructure);
    static bool is_a(StructureConstPtr const & structure);
    static bool is_a(PVStructurePtr const & pvStructure);
    static bool isCompatible(StructureConstPtr const & structure);
    static bool isCompatible(PVStructurePtr const & pvStructure);
    static NTScalarMultiChannelBuilderPtr createBuilder();

    bool attachTimeStamp(PVTimeStamp &pvTimeStamp) const;
    bool attachAlarm(PVAlarm &pvAlarm) const;
    PVStringPtr getDescriptor() const;
    PVStructurePtr getPVStructure() const;
    PVStructurePtr getTimeStamp() const;
    PVStructurePtr getAlarm() const;


    PVScalarArrayPtr getValue() const;
    template<typename PVT>
    std::tr1::shared_ptr<PVT> getValue() const;

    PVStringArrayPtr getChannelName() const;
    PVBooleanArrayPtr getIsConnected() const;
    PVIntArrayPtr getSeverity() const;
    PVIntArrayPtr getStatus() const;
    PVStringArrayPtr getMessage() const;
    PVLongArrayPtr getSecondsPastEpoch() const;
    PVIntArrayPtr getNanoseconds() const;
    PVIntArrayPtr getUserTag() const;
private:
    // ... remainder of class definition
}
```

where

getValue() Returns the `value` field.

getChannelName() Returns the `name` field. (Contains the name of each channel.)

getIsConnected() Returns the additional `isConnected` field. (Contains the connection state of each channel.) This is not an optional field of the type, but is commonly included.

getSeverity() Returns the `severity` field. (Contains the alarm severity of each channel.)

getStatus() Returns the `status` field. (Contains the alarm status of each channel.)

getMessage() Returns the `message` field. (Contains the alarm message of each channel.)

getSecondsPastEpoch() Returns the `secondsPastEpoch` field. (Contains the timeStamp secondsPastEpoch of each channel.)

getNanoseconds() Returns the `nanoseconds` field. (Contains the timeStamp nanoseconds of each channel.)

getUserTag() Returns the `userTag` field. (Contains the timeStamp userTag of each channel.)

and all other functions are described in the sections *Features common to all Normative Type Wrapper classes* and *Normative Type Property Features*.