Quick Guide : Adding Fields to Normative Types

This document serves as a quick guide on how to add extra fields to a normative type pvData structure.

Prerequisites:

Build of EPICS v3 and v4.

Familiarity with the EPICS v4 modules: pvDataCPP, normativeTypesCPP.

The pvData module allows for the creation of custom pvData structures with an arbitrary number of fields. The fields can be of any type defined by pvData, or a nested pvData structure. To allow interoperability between different code bases, a standardised definition of structures, normative types, also exists. These normative types are not defined as separate types from the pvData structures. Normative types are simply standardised specifications for pvData structures. These definitions are specified in the normative type documentation.

The documentation specifies that all normative types have certain fields that are required and allow for certain fields that are optional. What fields are required is dependent on the specific normative type. However every normative type is also allowed any arbitrary number of extra fields. These fields can be any pvData structure or pvData type.

The purpose of this document is thus to provide an example of how to add extra fields to a normative type structure. As mentioned above, this field can be any pvData type or pvData structure.

Instantiating a Normative Type pvData Structure

Every normative type has an associated builder class. For example, instances of 'ntScalar' (the scalar normative type) are allocated using the 'ntScalarBuilder' class. To obtain the builder class, use the associated 'createBuilder()' method.

For example:

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

The normative type is then defined by using the builder's 'add*()' methods.

For example:

```
PVStructurePtr pvStructure = ntScalarBuilder->
        value(pvDouble)->
        addAlarm()->
        addTimeStamp()->
        createPVStructure();
```

This pvData structure is now of the type ‘ntScalar’ where the scalar value is a double floating point value. The ‘value’ field is required, and its method is thus not prepended with ‘add’. The optional fields (often alarms, timestamps, and other associated metadata) are added to the structure using the ‘add*()’ methods.

Instantiating a Normative Type pvData Structure with Extra Fields

To add an extra field to a normative type pvData structure the same builder class is used as before. In this example, a nested pvData structure and a pvData type will be added as two extra fields in the normative type pvData structure.

Two instances of builder classes, ‘FieldCreate’ and ‘PVDataCreate’, are needed to define the nested structure. These singleton class instances are retrieved using their respective ‘get*()’ methods.

For example:

```
FieldCreatePtr fieldCreate = getFieldCreate();
PVDataCreatePtr pvDataCreate = getPVDataCreate();
```

FieldCreate is then used to define the nested structure. It’s important to note that in this example fieldCreate created an instance of a fieldBuilder object using the createFieldBuilder() method. This is done in-line because the fieldBuilder object is only going to be used once. When creating multiple structures it is better to create a single instance of a fieldBuilder object, and then use it through a pointer. Creating it in-line, as shown here, would be inefficient for creating multiple structures, because a new instance would be allocated for every structure. One instance of a field builder can create any arbitrary number of structures over its lifetime.

For example:

```
StructureConstPtr structure = fieldCreate->createFieldBuilder()->
        add(“a_long_integer”, pvLong)->
        add(“a_double”, pvDouble)->
        add(“are_we_having_fun_yet”, pvBoolean)->
        createStructure();
```

Use the PVDataCreate instance to create a PVStructure from the previously defined ‘structure’.

For example:

```
    PVStructurePtr nestedPVStructure =
            pvDataCreate->createPVStructure(structure);
```

To add the pvData type as an extra field request an instance of the builder class, ‘standardField’.

For example:

```
    StandardFieldPtr standardField = getStandardField();
```

Adding the type as an extra field can be done inline when creating the normative type pvData structure.

Creating the normative type with additional fields:

```
    PVStructurePtr pvStructure = ntScalarBuilder->
            value(pvDouble)->
            addAlarm()->
            addTimeStamp()->
            add( "nested_structure", nestedPVStructure->getField() )->
            add( "extra_double", standardField->scalar(
                    pvDouble, "value")->getField("value") )->
            createPVStructure();
```

<u>All of the code:</u>

```
    NTScalarBuilderPtr ntScalarBuilder = NTScalar::createBuilder();
    PVStructurePtr pvStructure = ntScalarBuilder->
            value(pvDouble)->
            addAlarm()->
            addTimeStamp()->
            createPVStructure();
    FieldCreatePtr fieldCreate = getFieldCreate();
    PVDataCreatePtr pvDataCreate = getPVDataCreate();
    StructureConstPtr structure = fieldCreate->createFieldBuilder()->
            add("a_long_integer", pvLong)->
            add("a_double", pvDouble)->
            add("are_we_having_fun_yet", pvBoolean)->
            createStructure();
```

```
PVStructurePtr nestedPVStructure =
        pvDataCreate->createPVStructure(structure);
StandardFieldPtr standardField = getStandardField();
PVStructurePtr pvStructure = ntScalarBuilder->
        value(pvDouble)->
        addAlarm()->
        addTimeStamp()->
        add( "nested_structure", nestedPVStructure->getField() )->
        add( "extra_double", standardField->scalar(
                pvDouble, "value")->getField("value") )->
        createPVStructure();
```

The resulting normative type will be of the structure:

```
epics:nt/NTScalar:1.0
        double value
        alarm_t alarm
                int severity
                int status
                string message
        time_t timeStamp
                long secondsPastEpoch
                int nanoseconds
                int userTag
        structure nested_structure
                long a_long_integer
                double a_double
                boolean are_we_having_fun_yet
        double extra_double
```

After the record has been created, and the code is running, using ‘pvinfo recordName’ or ‘pvget -r “” recordName’ will produce the structure above on the command line.

There is code available that demonstrates creating this exact record and hosting it on an EPICS v4 pvDatabase.