# EPICS pvDatabase and Normative Type Tutorial

Author: Evan Smith (SNS Intern)

## 1. Overview

This document serves as an introductory tutorial to the EPICS version 4 pvDatabase and normative type modules.

### A. Introduction:

In this tutorial a pvDatabase will be populated with pvRecords that contain normative type pvData structures. The demo will employ two programs, a client and a server. The client will operate on data that is hosted on the database, which is made accessible by the server.

### B. Prerequisites and Terminology:

-Knowledge of C++

-EPICS version 3 and EPICS version 4 build. See getting started with EPICS v3 and v4.

-Understanding of the EPICS build system.

-For brevity, normative types may be foreshortened to ‘nt’.

-Unless otherwise specified the term ‘database’ refers to the pvDatabase used in this tutorial.

-pvData structures are flexible data types. Normative types are standardised definitions for particular instances of pvData structures (to allow interoperability) but they are implemented with nothing more than pvData structures. pvRecords are “wrappers” that hold a pvStructure, i. e. a pvData structure. This allows records hosted on a pvDatabase to contain normative types and custom pvData structures alike.

### C. Modules Used:

**pvData**: The EPICS v4 type system, which describes and implements structured data.
**normativeTypes**: A set of standard pvData structures.

**pvAccess**: Support for connecting a client and server and for transporting pvData between client and server.
**pvaClient**: pvAccess is a callback based API. pvaClient is a synchronous interface for pvAccess and also provides a number of convenience methods.
**pvDatabase**: A memory resident database of pvRecords (memory resident structures that hold pvData) and a pvAccess channel provider for accessing the pvRecords.

### D. Code Repository:

https://github.com/es-sns121/ntDatabase-Demo/

## 2. Build

The EPICS build system, regardless of version, expects a shell variable to be set called 'EPICS_HOST_ARCH'. This defines the target architecture. For example, EPICS_HOST_ARCH might be set to "linux-x86_64". Setting this with an export command in a shell startup file will smooth things out down the line.

### A. EPICS v3

The first requirement is a build of EPICS version 3 (3.16 as of this writing). This is commonly referred to as 'epics base'. A download and a step by step guide for installation is available.

### B. EPICS v4

Once EPICS v3 has been installed, EPICS version 4 (4.6.0 as of this writing) needs to be downloaded and built according to the steps outlined in the installation guide. The guide specifies the steps necessary to build EPICS v4, however it references version 4.5.0. It is recommended to follow the steps as they are outlined, but with the latest available version (4.6.0 or greater) instead of 4.5.0.

### C. Creating a project directory:

#### a. Cannibalising an already existing project

Navigate to the EPICS v4 'exampleCPP' directory. A directory called 'ntDatabase' is going to be created by cannibalizing an already existing one. Copy the 'database' directory from the 'exampleCPP' directory and rename it to 'ntDatabase' in the new location. The directories 'db', 'dbd', 'ioc', and 'iocBoot' can be deleted. These directories would be populated if the pvaSrv module was

going to be used. This module allows an EPICS v4 server to host EPICS v3 records.

There are two Makefiles that need to be modified. The Makefile in the topmost directory of 'ntDatabase' can have the lines adding 'ioc' and 'iocBoot' directories to the directory list removed as those directories have been deleted because the tutorial does not use them.

The Makefile of the top most directory :

```
1  # Makefile at top of application tree
2
3  TOP = .
4  include $(TOP)/configure/CONFIG
5
6  DIRS += configure
7
8  DIRS += src
9  src_DEPEND_DIRS = configure
10
11 include $(TOP)/configure/RULES_TOP
12
13
```

The alterations to the second Makefile are more significant. Navigate to the ntDatabase/src directory.

All instances of 'example*' should be replaced with the corresponding name of the 'ntDatabase*' file. There should only be two production hosts (executable the makefile produces), 'ntDatabaseClient' and 'ntDatabaseMain'. The '*_LIBS' variable should define what libraries the production hosts depend on. There should be a single '*_LIBS' variable for each production host. To compile the headers and secondary source files as libraries, add all the secondary source files to their respective library variables. Then set the libraries own '*_LIBS' variable to the library's dependencies.

The lines regarding 'DBD' can be removed as they refer to the database files that define the structure of EPICS v3 records. These would be necessary if the pvaSrv module was being used.

The source directory makefile should look like this:

```
3  TOP=..
4
5  include $(TOP)/configure/CONFIG
6
7  #USR_CXXFLAGS += -std=c++0x
8  ntDatabaseSRC = $(TOP)/src
9
10 # Includes
11 INC += pv/ntDatabase.h
12 INC += ntScalarDemo.h
13 INC += ntDemo.h
14
15 # Lib
16 LIBRARY += ntDatabase
17 LIBSRCS += ntDatabase.cpp
18 LIBRARY += ntDemo
19 LIBSRCS += ntScalarDemo.cpp ntDemo.cpp
20 ntDatabase_LIBS += pvaClient pvDatabase pvAccess nt pvData Com
21 ntDemo_LIBS +=  ntDatabase pvaClient pvDatabase pvAccess nt pvData Com
22
23 # Server
24 PROD_HOST += ntDatabaseMain
25 ntDatabaseMain_SRCS += ntDatabaseMain.cpp
26 ntDatabaseMain_LIBS += ntDatabase
27 ntDatabaseMain_LIBS += pvaClient pvDatabase pvAccess nt pvData Com
28
29 # Client
30 PROD_HOST += ntDatabaseClient
31 ntDatabaseClient_SRCS += ntDatabaseClient.cpp
32 ntDatabaseClient_LIBS += ntDemo
33 ntDatabaseClient_LIBS += pvaClient pvAccess nt pvData ca Com
34
35 # Shared Library ABI version.
36 SHRLIB_VERSION ?= 1.0
37
38 include $(TOP)/configure/RULES
```

## a. Creating the project from scratch

To make a project directory from scratch there are two options. Option one, and recommended, is to utilize the EPICS build system.

**Option One:**

The system expects a specific directory structure. Topmost is the project directory. It's name is irrelevant to the build system but should be specific to whatever project is being built. In the topmost directory, two subdirectories are expected: a source directory and a configure directory. These directories will be referenced by the topmost makefile, which will be written next, so their names are up to the programmer's discretion. Traditionally they are named 'src' and 'configure'.

The topmost makefile is the same as before if we name the two sub directories 'src' and 'configure':

```
1  # Makefile at top of application tree
2
3  TOP = .
4  include $(TOP)/configure/CONFIG
5
6  DIRS += configure
7
8  DIRS += src
9  src_DEPEND_DIRS = configure
10
11 include $(TOP)/configure/RULES_TOP
12
13
```

The configure directory needs to be populated with the following files: CONFIG, CONFIG_SITE, Makefile, RELEASE.$HOST_ARCH.Common, RELEASE, RULES, RULES_DIR, and RULES_TOP.

These files should be copied from one of the EPICS v4 example directories. Note that the module directories' (pvData, pvAccess, etc.) RELEASE.linux-x86_64.Common file only contains paths to their own

specific dependencies. Note also that the target architecture listed in the RELEASE.*.Common file may be different depending on the specific system.

The files in the configure directory provide the build system with system specific information about the host system. The configure directory also holds the RULES* files that determine how the project is to be built.

RELEASE.*.Common files can also be generated using the genRelease.pl script located in '/epics-cpp-4.6.0/tools/'. The syntax for creating the script is as follows:

```
perl -CSD genRelease.pl -o [outputFile] -4 [epics_v4_path] -B [epics_base_path] \
     -h [Host_Arch] [dependencies]
```

'outputFile' allows you to select the name of the output file. The default filename is 'RELEASE.$(HOST_ARCH).Common'. The '-4' flag lets you designate the path to the epics v4 directory. The '-B' flag lets you designate the path to the epics v3 directory. Dependencies is an arbitrarily long list of epics modules to define pathways for.

For Example:

```
> pwd
/home/$USER/epics-cpp-4.6.0/

> perl -CSD tools/genRelease.pl  -o testRelease \
     -h linux-x86_64
     -4 /home/$USER/epics-cpp-4.6.0 \
     -B /home/$USER/epics-base-3.16 \
     exampleCPP pvDatabaseCPP pvaSrv \
     pvaClientCPP pvAccessCPP normativeTypesCPP \
     pvDataCPP pvCommonCPP
```

The above commands generated the following RELEASE file:

```
# testRelease
#     Generated by /home/$USER/epics-cpp-4.6.0/tools/genRelease.pl

EXAMPLE = /home/$USER/epics-cpp-4.6.0/exampleCPP
PVDATABASE = /home/$USER/epics-cpp-4.6.0/pvDatabaseCPP
PVASRV = /home/$USER/epics-cpp-4.6.0/pvaSrv
PVACLIENT = /home/$USER/epics-cpp-4.6.0/pvaClientCPP
PVACCESS = /home/$USER/epics-cpp-4.6.0/pvAccessCPP
```

```
NORMATIVETYPES = /home/$USER/epics-cpp-4.6.0/normativeTypesCPP
PVDATA = /home/$USER/epics-cpp-4.6.0/pvDataCPP
PVCOMMON = /home/$USER/epics-cpp-4.6.0/pvCommonCPP
EPICS_BASE = /home/$USER/epics-base-3.16
```

Once the configure directory is set up, the makefile for the source directory needs to be created.

The Makefile for the source directory will be the same as shown above where an existing directory will be cannibalised. Remember to change the include statements to include the configure directory with the appropriate name.

**Option Two:**

Option two for creating a project from scratch is to *not* use the EPICS build system. A single makefile will then build the project. For this, a single project directory need be made, but other sub directories can be added at the programmer's discretion. The makefile needs to include all of the EPICS modules of v3 and v4. It then needs to provide paths to the libraries and link the libraries for those modules. This is done using the -I (uppercase i), -L, and -l (lowercase L) flags respectively.

The single Makefile will end up looking something like this:

Include EPICS include directories.

```
top = ..

include $(top)/configure/config

bin = $(top)/bin

EPICS_INCLUDE += -I$(EPICS_V3_DIR)/include                  \
                                -I$(EPICS_V3_DIR)/include/compiler/gcc       \
                                -I$(EPICS_V3_DIR)/include/valgrind           \
                                -I$(EPICS_V3_DIR)/include/os/$(HOST_OS)      \
                                -I$(EPICS_V4_DIR)/pvAccessCPP/include        \
                                -I$(EPICS_V4_DIR)/pvDataCPP/include          \
                                -I$(EPICS_V4_DIR)/normativeTypesCPP/include  \
                                -I$(EPICS_V4_DIR)/pvaClientCPP/include       \
                                -I$(EPICS_V4_DIR)/pvDatabaseCPP/include      \
                                -I$(EPICS_V4_DIR)/pvCommonCPP/include

```

Include EPICS library directories and link EPICS libraries.

```
17
18 EPICS_LIBRARY += -L$(EPICS_V3_DIR)/lib/$(EPICS_HOST_ARCH) -lCom                    \
19                                 -L$(EPICS_V4_DIR)/pvDataCPP/lib/$(EPICS_HOST_ARCH) -lpvData         \
20                                 -L$(EPICS_V4_DIR)/pvAccessCPP/lib/$(EPICS_HOST_ARCH) -lpvAccess     \
21                                 -L$(EPICS_V4_DIR)/pvCommonCPP/lib/$(EPICS_HOST_ARCH) -lpvMB         \
22                                 -L$(EPICS_V4_DIR)/pvaClientCPP/lib/$(EPICS_HOST_ARCH) -lpvaClient   \
23                                 -L$(EPICS_V4_DIR)/pvDatabaseCPP/lib/$(EPICS_HOST_ARCH) -lpvDatabase \
24                                 -L$(EPICS_V4_DIR)/normativeTypesCPP/lib/$(EPICS_HOST_ARCH) -lnt
25
26 CPP_FLAGS = -Wall -g -lpthread -lm
27
```

Build the Client and Server.

```
27
28 all : client server
29
30 # Client Sources
31 clientSrc = ntDatabaseClient.cpp ntDemo.cpp ntScalarDemo.cpp
32 # Client Dependencies
33 clientDep = ntDemo.h ntScalarDemo.h $(clientSrc)
34
35 # Server Sources
36 serverSrc = ntDatabaseMain.cpp ntDatabase.cpp
37 # Server Dependencies
38 serverDep = pv/ntDatabase.h $(serverSrc)
39
40 client: $(clientDep)
41         mkdir -p $(top)/bin
42         g++ $(CPP_FLAGS) $(EPICS_INCLUDE) $(EPICS_LIBRARY) $(clientSrc) -o $(bin)/client
43
44 server: $(serverDep)
45         mkdir -p $(top)/bin
46         g++ $(CPP_FLAGS) $(EPICS_INCLUDE) $(EPICS_LIBRARY) $(serverSrc) -o $(bin)/server
47
48 .PHONY: clean
49 clean:
50         @printf "Cleaning binaries...\n"
51         rm -rf $(bin)/*
```

## 3. Server

The server program will be constructed first. There are three files that make up the server: a header file, a main source file, and a secondary source file. The header file will define the project's namespace and define a class, 'ntDatabase'. This class will have a single member function that will instantiate a database and populate it with records. The secondary source file will implement the ntDatabase class's single function which populates the database with records. These records will hold normative type pvData structures, one record for each normative type.

The main source file is where the database will be instantiated and a local access server started. It then waits for user input to exit.

## A. Header File : ntDatabase.h

The header file will define the project's namespace, which will be a sub-namespace of the 'epics' namespace. A single class, 'ntDatabase', will be defined. ntDatabase will have a single member function, 'create()', that will instantiate a pvDatabase and populate it with records. The create function will return void. The database will be accessed by a pointer, but it will be returned from another function called in the main source file.

```
1  #ifndef NTDATABASE_H
2  #define NTDATABASE_H
3
4  #ifdef epicsExportSharedSymbols
5  #       define  ntDatabaseEpicsExportSharedSymbols
6  #       undef   epicsExportSharedSymbols
7  #endif
8
9  #include <pv/pvDatabase.h>
10
11 #ifdef ntDatabaseEpicsExportSharedSymbols
12 #       define epicsExportSharedSymbols
13 #       undef  ntDatabaseEpicsExportSharedSymbols
14 #endif
15
16 #include <shareLib.h>
17
18 namespace epics { namespace ntDatabase {
19
20         class epicsShareClass NTDatabase {
21                 public:
22                         static void create();
23         };
24
25 }}
26
27
28 #endif /* NTDATABASE_H */
```

## B. Secondary Source File : ntDatabase.cpp

The secondary source file is where the 'epics::ntDatabase::create()' method and its associated helper function will be implemented. This file implements two functions. The aforementioned 'create' method, and a 'createScalarRecords' helper function. The create method can be logically broken up into two parts.

The first part of the 'create()' function will create the scalar normative type records and scalar array normative type records. These records will hold the normative types 'scalar' and 'scalarArray'. This can be done with a generic function, 'createScalarRecords', that handles any scalar type and its associated array record (credit for this idea goes to Marty Kraimer at BNL). For example, the 'createScalarRecords' method will create a pvRecord of type PVDouble and another that's an array of PVDoubles.

```
55  // Generic record creation function.
56  static void createScalarRecords(
57          PVDatabasePtr const &master,
58          ScalarType scalarType,
59          string const &recordNamePrefix)
60  {
61          string recordName = recordNamePrefix;
62
63          NTScalarBuilderPtr ntScalarBuilder = NTScalar::createBuilder();
64
65          // Create the pvStructure to be inserted into the record.
66          PVStructurePtr pvStructure = ntScalarBuilder->
67                  value(scalarType)->
68                  addAlarm()->
69                  addTimeStamp()->
70                  createPVStructure();
71
72          // Create the record and attempt to add it to the database.
73          PVRecordPtr pvRecord = PVRecord::create(recordName, pvStructure);
74
75          bool result = master->addRecord(pvRecord);
76          if (!result) cerr << "Failed to add record " << recordName << " to database\n";
77
78          recordName += "Array";
79
80          // Create the pvStructure for the array type to be inserted into the record.
81          NTScalarArrayBuilderPtr ntScalarArrayBuilder = NTScalarArray::createBuilder();
82          pvStructure = ntScalarArrayBuilder->
83                  value(scalarType)->
84                  addAlarm()->
85                  addTimeStamp()->
86                  createPVStructure();
87
88          pvRecord = PVRecord::create(recordName, pvStructure);
89
90          result = master->addRecord(pvRecord);
91          if (!result) cerr << "Failed to add record " << recordName << " to database\n";
92
93          return;
94  }
```

The second part of the 'create()' function will create the rest of the normative type records. The method to create and add these records to a database follows a general pattern. Each normative type has an associated builder class. For example, the normative type 'NTEnum' has the associated builder class 'NTEnumBuilder'. To get an instance of a normative type, request a pointer to the singleton builder class associated to that normative type. Then use this builder to return an instance of the normative type. Once the normative type is instantiated, it will then be 'wrapped' in a pvRecord. This pvRecord will then be added to the database.

```
96  // Creates and adds records to database.
97  void NTDatabase::create()
98  {
99          bool result(false);
100
101         // Get the database hosted by the local provider.
102         PVDatabasePtr master = PVDatabase::getMaster();
103
104         // Create string and string array records.
105         createScalarRecords(master, pvString, "string");
106         // Create numeric type and numeric type array records.
107         createScalarRecords(master, pvShort, "short");
108         createScalarRecords(master, pvInt, "int");
109         createScalarRecords(master, pvLong, "long");
110         createScalarRecords(master, pvDouble, "double");
111
112         /* ================================================= */
113         // Create a NTEnum pvrecord.
114
115         NTEnumBuilderPtr ntEnumBuilder = NTEnum::createBuilder();
116
117         PVStructurePtr pvStructure = ntEnumBuilder->
118                 addAlarm()->
119                 addTimeStamp()->
120                 createPVStructure();
121         // Create the choices vector for the enum.
122         shared_vector<string> choices(2);
123         choices[0] = "zero";
124         choices[1] = "one";
125         // Get the structure's array and replace it with the newly created vector.
126         PVStringArrayPtr pvChoices = pvStructure->getSubField<PVStringArray>("value.choices");
127         pvChoices->replace(freeze(choices));
128
129         result = master->addRecord(PVRecord::create("enum", pvStructure));
130         if (!result) cerr << "Failed to add enum record\n";
131
```

Only the first of the other normative types is shown. The methodology for adding other types is very similar. It follows the same pattern of requesting a normative type builder, creating a pvStructure, and then creating the pvRecord. Creating all other normative types is shown in the source code included with this tutorial (ntDatabase.cpp).

As an aside, it is possible to add any arbitrary number of fields to a normative type. In review, a normative type is simply a specification for a pvData structure. There are required fields and optional fields, but the specification also allows for the addition of any number of additional fields. These additional fields can be added using the add() method. Though only the add() method for NTScalarBuilder is linked, all normative type builder classes have this method. The add method takes a string, the field name, and a field pointer (FieldConstPtr). The field pointer for any pvData structure can be retrieved using the getField() method of the structure. This allows for any custom or standard pvData structure to be added as an additional field to a normative type. Here, is a more in depth guide to adding fields to normative types.

## C. Main Source File : ntDatabaseMain.cpp

The main source file is where 'epics::ntDatabase::create()' is called. The local access server that will provide access to the database is started afterwards. The pvAccess protocol is being used in this tutorial so the 'startPVAServer()' method is called to start the access server.

```
66
67          // Get the master database maintained by the local channel provider.
68          PVDatabasePtr master = PVDatabase::getMaster();
69
70          // Get the local channel provider.
71          ChannelProviderLocalPtr cpLocal = getChannelProviderLocal();
72
73          // Create the normative type database that is defined locally in pv/ntDatabase.h
74          NTDatabase::create();
75
76          // After the records are added to the database, start the server.
77          ServerContext::shared_pointer pvaServer =
78                  startPVAServer("local", 0, true, true);
79
```

First, a pointer to the database is needed to populate it. The database "master" is maintained by the local channel provider. A pointer to the local channel provider will also be obtained so that the server can be shut down smoothly upon exit. After that, the database is populated with records by calling the 'NTDatabase::create()' method. Once the local database is populated with records, start the pvAccess server that will provide access to the database.

```
89
90          // Clear the pointer.
91          master.reset();
92
93          // Wait to die.
94          string input;
95          while (true)
96          {
97                  cout << "Type exit to stop: \n";
98                  getline(cin, input);
99                  if (input.compare("exit") == 0)
100                 {
101                         break;
102                 }
103         }
104
105         // Clean up so that we can exit cleanly.
106         pvaServer->shutdown();
107         pvaServer->destroy();
108         cpLocal->destroy();
109
110         return 0;
111 }
```

After the server is started it will wait until the user types “exit”. At that point it will clean up and exit. Notice that the database only exists so long as this program runs. It is memory resident in this process’s address space.

## 4. Client

The client code is made fairly simple by using the pvaClient module. This is a synchronous wrapper around the callback based pvAccess module. However, things will be complicated somewhat by having numerous demonstration functions. These functions will demonstrate how to use the pvaClient API to access the normative type pvRecords hosted on the pvDatabase. The client code will demonstrate how to use each normative type.

The client is made up of five files: a main source file, two header files, and two secondary source files that implement demonstration functions defined in the respective header files.

The main source file handles creation of the pvaClient object which will be used to communicate to the server. It will then call the demonstration functions. Each demonstration

function follows a pattern. First, a pvAccess channel is opened for communication to the pvRecord. The channel name will be the pvRecord name. After the channel is connected, a 'putGet' object is created. This will allow data to be put to and gotten from the record. The 'putGet' structure maintains local buffers that contain copies of the structure hosted in the pvRecord. These copies must be manually updated by calling two methods. The 'put' copy is requested using 'getPutData()' and then is used to write to the record. After writing, the 'putGet()' method is called to flush the changed bitsets to the server. This will update the record on the server. The 'getGetData()' method is then called to request the most recent copy of data from the record on the server. The data retrieved from the record can now be compared to the data written to the record. This should demonstrate all that is needed to know for how to write to and read from pvRecords that contain normative types.

## A. Main Source File

The main source code for the client is fairly simple. First the singleton pvaClient class will be retrieved by using the PvaClient::get("pva") function. The argument "pva" is passed to select the pvAccess protocol. This is the default protocol, but the channel access protocol can be selected instead of the pvAccess protocol. A debug flag can also be set in the 'pvaClient' object. This will print out useful information as to what is going on internally in the pvaClient object. The main function parses some command line arguments to verify if the user wants this debug option to be enabled or not.

The demonstration functions are called from a wrapper function, 'demoRecord()'. It parses the arguments it was passed and then calls the requested demonstration function. It does this by looking through a map keyed on the demonstration function names. The values held in the map are function pointers to the respective demonstration functions. This will be discussed further when the secondary source files are described. 'demoRecord()' then calls the demonstration function and prints the results using the 'printResult()' method.

The 'record_types' array in the main client loop is an array of strings, initialized to all the record types of the records on the database. The channel names for the records are set to be the same as their record type in ntDatabase.cpp. The loop goes through the array calling 'demoRecords()' on each record type.

```
80          cout << "ntDatabase Client\n";
81
82          string record_types[] = {
83                                          "string", "short", "int", "long", "double",
84                                          "stringArray", "shortArray", "intArray",
85                                          "longArray", "doubleArray",
86
87                                          "enum", "matrix", "uri", "name_value", "table",
88                                          "attribute", "multi_channel"};
89
90          int number_of_record_types = 16;
91
```

```
89
90          int number_of_record_types = 16;
91
92          try {
93
94                  PvaClientPtr pvaClient = PvaClient::get("pva");
95
96                  cout << "debug : " << (debug ? "true" : "false") << endl;
97
98                  if (debug) PvaClient::setDebug(true);
99
100                 // seed rand for the generator functions in the demo code.
101                 srand(time(NULL));
102
103                 string channel_name;
104
105                 // Demo the nt records.
106                 for (int i = 0; i < number_of_record_types; ++i) {
107
108                         channel_name = record_types[i];
109
110                         demoRecord(verbosity, pvaClient, channel_name);
111
112                         channel_name.clear();
113                 }
114
115         } catch (std::runtime_error e) {
116                 cerr << "exception: " << e.what() << endl;
117                 return -1;
118         }
119
120         return 0;
121 }
```

After calling all of the demonstration functions and printing the results, the client exits.

## B. Header Files - ntDemo.h & ntScalarDemo.h

The header files for the client define the function APIs that will be implemented in the client secondary source files.

### a. ntDemo.h

ntDemo.h defines the ‘printResult()’ and ‘demoRecord()’ functions. It instantiates and initializes a map of (string / function pointer) pairs. The ‘printResult()’ function prints a formatted message that displays the result of the demonstration function. ntDemo.h also defines a number of demonstration functions for the non-scalar normative type records.

```
45  void printResult(const bool &result, const string &channel_name);
46
47  int demoRecord(
48          bool verbosity,
49          PvaClientPtr pva,
50          string const & channel_name);
51
52  bool demoEnum(
53          bool verbosity,
54          PvaClientChannelPtr channel);
55
56  bool demoMatrix(
57          bool verbosity,
58          PvaClientChannelPtr channel);
```

These are not all of the function definitions, but represent the general pattern they conform to. The verbosity boolean indicates how much information is printed to stdout. The PvaClientPtr is used to communicate to the server. The demonstration functions are passed a PvaClientChannelPtr, which is set up in demoRecord. They use this channel to communicate to the records on the server.

## b. ntScalarDemo.h:

ntScalarDemo.h defines a number of demonstration functions for the scalar normative type records. It also defines three generator functions that produce random integer values, double precision floating point values, and alphanumeric strings. These serve little practical use as they are poor examples of random data generation. They only serve to generate pseudo random input that can be written to records.

```
37  // Generates a string of "random" length and "random" alpha-numeric content.
38  string genString();
39
40  bool demoString(
41          bool verbosity,
42          PvaClientChannelPtr channel);
43
44  bool demoStringArray(
45          bool verbosity,
46          PvaClientChannelPtr channel);
47
48  // Generates a "random" integer in the range 0 to high.
49  long genInt(long high);
50
51  bool demoShort(
52          bool verbosity,
53          PvaClientChannelPtr channel);
```

These are not all of the function definitions but demonstrate the general pattern they conform to.

## C. Secondary Source Files - ntDemo.cpp & ntScalarDemo.cpp

### a. ntDemo.cpp:

The print result function is trivial. It prints the result and the channel name, then interprets if it was successful or not.

```
44  /* Formatted print function for demo function results */
45  void printResult(const bool &result, const string &channel_name) {
46
47          if (result) {
48                  cout << channel_name << " record demo successful\n";
49          } else {
50                  cout << channel_name << " record demo unsuccessful\n";
51          }
52
53          return;
54  }
```

The demoRecord function instantiates a map keyed on channel names and whose values are function pointers to the respective demonstration functions. The

'first_call' boolean allows only instantiating the static variables once.

```
62          bool result(false);
63
64          static bool first_call = true;
65
66          /* map of function pointers keyed on channel name */
67          static map<string, bool (*) (bool, PvaClientChannelPtr)> functions;
68
69          map<string, bool (*) (bool, PvaClientChannelPtr)>::iterator it;
70
71          if (true == first_call) {
72          /* init functions map */
73                  first_call = false;
74                  functions["string"] = &demoString;
75                  functions["stringArray"] = &demoStringArray;
76                  functions["short"] = &demoShort;
77                  functions["shortArray"] = &demoShortArray;
78                  functions["int"] = &demoInt;
79                  functions["intArray"] = &demoIntArray;
80                  functions["long"] = &demoLong;
81                  functions["longArray"] = &demoLongArray;
82                  functions["double"] = &demoDouble;
83                  functions["doubleArray"] = &demoDoubleArray;
84                  functions["enum"] = &demoEnum;
85                  functions["matrix"] = &demoMatrix;
86                  functions["uri"] = &demoURI;
87                  functions["name_value"] = &demoNameValue;
88                  functions["table"] = &demoTable;
89                  functions["attribute"] = &demoAttribute;
90          }
```

The 'demoMultiChannel' function could not be added to the map, as it has a different function definition than the rest of the demo functions. It is handled with a simple 'if' statement.

```
92          if (channel_name.compare("multi_channel") == 0) {
93                  result = demoMultiChannel(verbosity, pva, channel_name);
94                  printResult(result, channel_name);
95                  return 0;
96          }
97
```

The function is then searched for in the map based on the channel name that it is passed. If a function is found, then it is called and its result is printed.

```
98          it = functions.find(channel_name);
99
100         if (functions.end() == it) {
101
102                 cerr << "Channel '" << channel_name << "' not recognised.\n";
103                 return 1;
104
105         } else {
106
107                 PvaClientChannelPtr channel = pva->channel(channel_name);
108
109                 if (channel) cout << "\nChannel \"" << channel_name << "\" connected succesfully\n";
110                 else
111                         return 1;
112
113                 result = (it->second)(verbosity, channel);
114                 printResult(result, channel_name);
115
116         }
117
118         return 0;
119
120 }
```

A PvaClientChannel object is created to allow communication with the record held on the server. Remember, to open up a channel to a record, the channel name is the record's name. It's always a good idea to make sure the channel actually connected. This will be very helpful for debugging. There is no need to manually call the connect() and waitConnect() methods. They are automatically called when the channel object is created.

There are numerous demonstration functions, but only the 'NTTable' demo function will be gone over. The techniques used in it apply to all of the normative types and it demonstrates almost all of the ways a normative type can be accessed.

```
350         bool result(true);
351
352         // Create putGet to read and write to/from record.
353         PvaClientPutGetPtr putGet = channel->createPutGet("");
354         PvaClientPutDataPtr putData = putGet->getPutData();
355         PvaClientGetDataPtr getData = putGet->getGetData();
356
```

Create a ‘putGet’ object. This is the data structure that will allow writing to and reading from the record. There are ‘get’ and ‘put’ objects as well, but the ‘putGet’ allows instantiating a single object to complete the same task. It’s important not to forget to pass the “” empty string to the ‘createPutGet()’ method. If it is not passed, the default string is “value, alarm, timeStamp”, which tells the ‘putGet’ to retrieve those fields alone from the record, and not the entire structure. Later, when the ‘getGetData/getPutData’ methods are called, an ‘invalid pvRequest’ exception will be thrown if the “” is not passed.

The ‘putGet’ object works by maintaining two local buffers. The first buffer is the ‘get’ buffer. It maintains the most recent copy of the pvRecord from the server. The second buffer is the ‘put’ buffer. It maintains a local copy of the pvRecord. After writing to the ‘put’ buffer, the altered bitsets must be flushed to the server (a bitset is an array of bits that represent what fields in a record have been changed. The pvAccess protocol only transmits the changed data. This is more efficient than transmitting the entire record on every read or write). The server will then update the pvRecord based on the changed information.

After setting up the ‘putGet’, the data that is to be written to the record needs to be produced.

Firstly, the normative type table is a structure with scalar arrays and an array of strings whose contents are labels to the scalar arrays. The scalar arrays should be thought of as the columns of the table. In this example the table has three columns. They are: questions, answers, and recommendations. All are arrays of strings. The values of these string arrays are hard coded in. Using the generateString() function here would make it difficult to see the structure of the record because the strings are not consistent.

Before the function is reviewed, a brief introduction is required. The EPICS library makes use of smart pointers. The shared_vector is a typedef of a vector that uses smart pointers. If you aren’t familiar with them, here’s an excellent introduction to them. For the most part the shared_vectors operate just as the normal STL vectors do. The one caveat is that the pvRecords use const data types in their shared_vectors (ex. shared_vector<const int> my_vec). This means

their contents cannot be altered after instantiation. So to work around this, a sacrificial 'non-const' vector is filled with contents, and then the 'freeze()' method is used to move the contents out of the sacrificial vector and into a 'const-vector'. This allows the contents in the record to be altered because the 'replace()' method can be used to replace the record's 'const vector'.

In this example the 'data' vector is reused three times as the sacrificial vector. The 'freeze()' method is then used, moving the contents from the sacrificial data vector to the 'const vector'.

```
357         // Create the questions vector
358         shared_vector<string> data;
359         data.push_back("Why are we here?");
360         data.push_back("How are we to be happy?");
361         data.push_back("Whats the meaning to life?");
362         data.push_back("Whats the answer to the ultimate question of life, the universe, and everything?");
363         shared_vector<const string> questions(freeze(data));
364
365         // Create the answers vector.
366         data.push_back("42");
367         shared_vector<const string> answers(freeze(data));
368
369         // Create the recommendations vector
370         data.push_back("Keep calm.");
371         data.push_back("Always carry a towel.");
372         data.push_back("Drink heavily and read the guide.");
373         shared_vector<const string> recommendations(freeze(data));
```

Once the 'const vectors' are filled with the desired contents, they are ready to be written to the record. This is done by using the 'replace()' method that will set the record's internal pointer to its 'const vector' to the new 'const vector'. In this way 'const' arrays can be written to pvRecords.

```
375         // Write the vectors to the table record. These consitute the tables columns
376         putData->getPVStructure()->getSubField<PVStringArray>("value.questions")->replace(questions);
377         putData->getPVStructure()->getSubField<PVStringArray>("value.answers")->replace(answers);
378         putData->getPVStructure()->getSubField<PVStringArray>("value.recommendations")->replace(recommendations);
379         putGet->putGet();
380
```

To select the desired array to replace, first request the PVStructure of the record. This will return a pvData structure that contains all the fields in the record. Then select the appropriate subfield with the 'getSubField<>()' method. Once the subfield is selected, use the 'replace()' method to swap out the vectors.

Having changed the vectors, it's now time to flush the changed bitsets to the server. This is done by calling the 'putGet()' method of the 'putGet' object.

```
381         putGet->getGetData();
382         shared_vector<const string> labels
383                 = getData->getPVStructure()->getSubField<PVStringArray>("labels")->view();
384
385         shared_vector<const string> questions_read
386                 = getData->getPVStructure()->getSubField<PVStringArray>("value.questions")->view();
387
388         shared_vector<const string> answers_read
389                 = getData->getPVStructure()->getSubField<PVStringArray>("value.answers")->view();
390
391         shared_vector<const string> recommendations_read
392                 = getData->getPVStructure()->getSubField<PVStringArray>("value.recommendations")->view();
393
```

To read the data from the record, first request the most recent copy of the record from the server. This is done using the 'getGetData()' method. There are two 'get' in the name because there is also a 'getPutData()' method that returns the local copy of the 'put' data. Once the most recent copy is attained, use the same pattern of requesting a PVStructure and then searching for the appropriate sub field. The 'view()' method will return the 'const vector' held in the pvRecord.

Writing arrays of data to a normative type pvRecord and then reading the data back from the record always follows this general pattern. Accessing arrays of data from any other pvData structure follows the same methodology.

For more examples, look at the source file ntDemo.cpp.

## b. ntScalarDemo.cpp:

**Scalar Demo:**

The ntScalar demo functions follow much the same pattern as discussed above with the table demo function. The long int demonstration function will be gone over. All of the scalar demonstration functions are near identical save for the types (Frustratingly, the author couldn't quite get a template version of the demo function working).

```
266  bool demoLong(
267          bool verbosity,
268          PvaClientChannelPtr channel)
269  {
270          bool result(true);
271
272          PvaClientPutGetPtr putGet = channel->createPutGet("");
273          PvaClientPutDataPtr putData = putGet->getPutData();
274          PvaClientGetDataPtr getData = putGet->getGetData();
```

The same steps are taken as described above in the table demo function for creating a 'putGet' object. These steps can be copied exactly for this function.

```
275
276          long write = genInt(INT_MAX);
277
278          putData->getPVStructure()->getSubField<PVLong>("value")->put(write);
279          putGet->putGet();
280
281          putGet->getGetData();
282
283          long read = getData->getPVStructure()->getSubField<PVLong>("value")->get();
284
285          if(verbosity)
286          {
287                  cout << setw(20) << "Write Long: " << write << "\n";
288                  cout << setw(20) << "Read Long: " << read << "\n\n";
289          }
290          if (write != read)
291                  result = false;
292
293          return result;
294  }
```

Dealing with single scalar values, as opposed to arrays, simplifies things. The steps are largely the same except the values can actually be read and written to directly because they are not 'const' qualified. The 'put()' method of the selected subfield will write a value to the scalar. Use the 'putGet()' method to flush the changed bitsets to the server. To read, request the newest copy of the data from the server using the 'getGetData()' method. Then use the 'get()' method of the selected subfield. It will return a copy of the scalar value held in the record.

**Scalar Array Demo:**

The ntScalarArray demo functions follow much the same pattern as the

ntTable demo did. The long int array demo function will be examined here.

```
296  bool demoLongArray(
297          bool verbosity,
298          PvaClientChannelPtr channel)
299  {
300          bool result(true);
301
302          PvaClientPutGetPtr putGet = channel->createPutGet("");
303          PvaClientPutDataPtr putData = putGet->getPutData();
304          PvaClientGetDataPtr getData = putGet->getGetData();
305
```

The same steps are taken as described above in the table demo function for creating a ‘putGet’ object. These steps can be copied exactly for this function.

```
306          // Number of longs in array is between 20 and 30
307          int num = (rand()%10) + 20;
308
309          shared_vector<long> data(num);
310
311          for (int i = 0; i < num; ++i)
312                  data[i] = genInt(INT_MAX);
313
314          shared_vector<const long> write(freeze(data));
315          // the data vector is now empty.
316
317          putData->getPVStructure()->getSubField<PVLongArray>("value")->replace(write);
318          putGet->putGet();
```

To write to the record, fill the sacrificial vector with the desired contents, and use the ‘freeze()’ method to move the contents into the ‘const vector’. Then use the ‘replace()’ method to write the vector to the local record. Flush the changed bitsets to the server with the ‘putGet()’ method. This will update the record hosted on the database.

```
320          putGet->getGetData();
321
322          // Read the data stored in the record.
323          shared_vector<const long> read;
324          read = getData->getPVStructure()->getSubField<PVLongArray>("value")->view();
325
```

Reading is the same as before. Request the latest copy of the record from

the database using the ‘getGetData()’ method. Then select the desired subfield, and use the ‘view()’ method to get the ‘const vector’.

The rest of the scalar demonstrations follow these exact patterns. For more examples look at the source file ntScalarDemo.cpp.

## 5. Conclusion

While not all inclusive, this guide has hopefully provided a good foundation for reading and writing data to and from pvRecords that contain normative type pvData structures.

For further reading, the pvData, normativeType, pvaClient and pvDatabase documentation all provide in depth explanations and details about the topics covered. There are also several other projects in the linked github repository. These projects are not guaranteed to work flawlessly, and they come with no documentation other than the README.md and the comments held within the source files. Regardless, the may provide as useful examples.