Channel Access Portable Server:

Reference Guide

Philip Stanley

November 1997 (Draft)
EPICS Release 3.13

Copyright

Initial development by:

The Controls and Automation Group (AOT-8),
Ground Test Accelerator,
Accelerator Technology Division,
Los Alamos National Laboratory.

Co-developed with:

The Controls and Computing Group,
Accelerator Systems Division,
Advanced Photon Source,
Argonne National Laboratory.

CHANNEL ACCESS PORTABLE SERVER: 1

REFERENCE GUIDE 1

Introduction 4

CLASS: caServer 8

CLASS: casPV 15

CLASS: casChannel 25

CLASS: casAsyncIO 32

CLASS: casAsyncReadIO 35

CLASS: casAsyncWriteIO 39

CLASS: casAsyncPVExistIO 43

CLASS: casAsyncCreatePVIO 47

CLASS: gdd 51

CLASS: gddAtomic 74

CLASS: gddScalar 77

CLASS: gddContainer 79

CLASS: aitString 81

CLASS: aitTimeStamp 86

CLASS: osiTime 91

CLASS: osiTimer 99

CLASS: gddAppFuncTable<PV> 105

Introduction

This reference covers the classes in the Portable Server's interface and the relevant public or protected members of each class. No private members are explained here because such members are generally not to be accessed by server tools. In addition, the `gdd` and `osiTimer` classes as well as the `gddAppFuncTable<PV>` template is covered, though not all of the public members will be covered since.

Common Types

These are some of the types used by the server library and its interface. Many of them are `typedefs`; some are classes. The classes are included here so that the programmer doesn't confuse simple defined types with class objects or vice versa, i.e., doesn't take `gdd` or `casCtx` to be simple `typedefs`.

```
typedef Type aitInt8
typedef Type aitUint8
typedef Type aitInt16
typedef Type aitUint16
typedef aitUint16 aitEnum16
typedef Type aitInt32
typedef Type aitUint32
typedef Type aitFloat32
typedef Type aitFloat64
typedef aitUint32 aitIndex
typedef void* aitPointer
typedef aitUint32 aitStatus;
```

Here, `Type` will vary according to the architecture of the EPICS buildÄsun4, Linux, WIN32, hkv2f, etc. For example, `aitInt8` should correspond to the eight-bit signed integer type on an architecture, often type `char`. After the EPICS build, aitTypes.h should indicate the appropriate types.

```
typedef enum {aitFalse=0, aitTrue} aitBool;
```

This is an architecture-independent boolean enumeration returned by some of the virtual functions from the server interface. Defined in aitTypes.h.

```
typedef struct {
   char fixed_string[AIT_FIXED_STRING_SIZE];
} aitFixedString;
```

Not used by the server library. Instead the server library uses `aitString`, a class, to represent strings. Maybe useful for non-C++ applications. Defined in aitTypes.h.

```
typedef struct {
   aitUint32 tv_sec;
   aitUint32 tv_nsec;} aitTimeStamp;
```

This is the architecture-independent time stamp structure. Used by `osiTime` and other classes. The first member is seconds; the second, nanoseconds. Defined in aitTypes.h.

```
class aitString;
```

This is a class whose declaration appears in aitHelpers.h which is included in aitTypes.h and so doesn't need to be included. The server library uses it to represent strings.

```
typedef enum {
   aitEnumInvalid=0, aitEnumInt8,  aitEnumUint8,
   aitEnumInt16, aitEnumUint16, aitEnumEnum16,
   aitEnumInt32, aitEnumUint32, aitEnumFloat32,
   aitEnumFloat64, aitEnumFixedString, aitEnumString,
   aitEnumContainer } aitEnum;
```

This is an enumerated type. Each enumerator corresponds to an architecture-independent type except for `aitEnumContainer` which is used to indicate a container object such as `gddContainer` and except for `aitEnumInvalid` which indicates an invalid type.

```
typedef union {
   aitInt8 int8; aitUint8 Uint8;
   aitInt16 Int16; aitUint16 Uint16;
   aitEnum16 Enum16; aitInt32 Int32;
   aitUint32 Uint32; aitFloat32 Float32;
   aitFloat64 Float64; aitIndex Index;
   aitPointer Pointer;  aitFixedString* FString;
   aitUint8 Dumb1[sizeof(aitString)]; // aitString
   aitUint8 Dumb3[sizeof(aitTimeStamp)];// timestamp
} aitType;
```

This is a union of all architecture-independent types. Note how `aitString` and `aitTimeStamp` are included.

```
enum pvExistReturnEnum {pverExistsHere,
pverDoesNotExistHere,
        pverAsyncCompletion};
```

This is an enumerated type used to communicate the existence/non-existence of a PV between the server library and the pvExistTest() function. The third enumerator indicates that the server tool wishes to determine the existence of the PV later. This enumerated type is actually used by the pvExistReturn class, which has a member of type `pvExistReturnEnum`.

```
class gdd;
```

This is the *Data Descriptor* class used extensively by the server library to represent data. The actual data is represented by an `aitType` member (see above). For scalar values, the corresponding union member is used. For instance, if the value is of type `aitInt8`, the member used is `Int8`. For arrays, the `Pointer` member of the `aitType` union points to the array. The `gdd` class is a base class for `gddScalar`, used only for scalar values, `gddAtomic`, used for array values, and `gddContainer`, used to contain other `gdd` objects.

```
class pvExistReturn;
```

The `pvExistReturn` class is used for communication between the server tool's `pvExistTest()` function and the server library. `pvExistTest()` contains a member of type `pvExistReturnEnum`, whose three enumeratorsÄ`pverExistsHere`, `pverDoesNotExistHere`, and `pverAsyncCompletion`Äindicate that `pvExistTEst()` found the PV in this server tool, didn't find the PV, or wishes to determine the existence of the PV asynchronously.

Although strictly speaking the return value of the `pvExistTest()` function is a `pvExistReturn` object, C++ allows a function to initialize and return an object indirectly. Thus, `pvExistTest()` can simply return one of the three possible values of `pvExistReturnEnum`, in which case a `pvExistReturn` object will be returned to the server library initialized with the specified enumerator.

```
class casCtx;
```

This is the *Channel Access Server Context* Class. The server library uses it mostly for asynchronous completion and access control, passing it to many virtual functions in the server interface. However, a server tool should almost never use it directly, but should only pass it to the appropriate constructors when doing asynchronous completion and access control, i.e., when dealing with the asynchronous IO classes and the `casChannel` class.

```
class casEventMask;
```

This is the *Channel Access Server Event* Mask class. `casEventMask` objects can be defined to represent different types of events. The server library allows a server tool to create any number of different types of events, each represented by its `casEventMask` object. However, the three main event types are initialized in the caServer base class. These objects are part of the `caServer` class. The unary operators `|` and `&` can be used with `casEventMask` objects as well as the `==` and `!=` binary operators.

```
typedef aitUint32 caStatus
```

This is the type of the status codes returned by the interface's functions to the server library.

```
#define S_casApp_success 0
#define S_casApp_noMemory (M_casApp | 1)
#define S_casApp_pvNotFound (M_casApp | 2)
#define S_casApp_badPVId (M_casApp | 3)
#define S_casApp_noSupport (M_casApp | 4)
#define S_casApp_asyncCompletion (M_casApp | 5)
```

```
#define S_casApp_badDimension (M_casApp | 6)
#define S_casApp_canceledAsyncIO (M_casApp | 7)
#define S_casApp_outOfBounds (M_casApp | 8)
#define S_casApp_undefined (M_casApp | 9)
```

These are the status codes returned to the server library by those server tool functions that return status codes. The value of `M_casApp` may be architecture dependent.

CLASS: caServer

Declared: casdef.h

The `caServer` class is the basic class for a server tool.

Destruction:

It is the server tool's responsibility to perform the necessary cleanup prior to an object's destruction. A virtual destructor, `~caServer()`, performs any cleanup necessary for the base class and its internals.

A virtual destructor, recall, is not inherited in C++. Instead, when a base class declares a destructor to be virtual, then all destructors in the derivation hierarchy are virtual. Virtual destructors are different than other virtual functions in that the virtual function in the base class, rather than providing a default function, guarantees instead an order of execution for the destructors in the derivation hierarchy. Thus, when a server tool derives a class from `caServer`, if the class provides a destructor, that destructor is called when a class object goes out of scope or, for free-store objects, when `delete` is applied to the free-store object. After the derived class' destructor is called, the base class virtual destructor is called.

Public Member Functions:

Non-Virtual

```
caServer (unsigned pvCountEstimate=1024u);
casEventMask registerEvent (const char *pName);
void setDebugLevel (unsigned level);
unsigned getDebugLevel ();
```

Virtual

```
virtual ~caServer();
virtual void show (unsigned level);
virtual pvExistReturn pvExistTest (const casCtx &ctx,
                        const char *pPVName)=0;
virtual casPV *createPV (const casCtx &ctx,
                    const char *pPVName)=0;
```

Public Members

```
const casEventMaskvalueEventMask;
const casEventMasklogEventMask;
const casEventMaskalarmEventMask;
```

Name:

`caServer()`

Synopsis:

```
#include <casdef.h>
caServer (unsigned pvCountEstimate=1024u);
```

Description:

`caServer()` is the class constructor. Its only argument `pvCountEstimate`, is simply an unsigned integer representing a rough estimate of the number of PVs that will be attached to the server tool. It is used by the server library to create a hash table. If the number of PVs actually attached to the server tool exceeds this number, a new hash table is created that is large enough to include the new PV as well.

The default for the argument is 1,024.

Name:

`registerEvent()`

Synopsis:

```
#include <casdef.h>
casEventMask registerEvent (const char *pName);
```

Description:

`registerEvent()` allows a server tool to register an event type with the server library. It accepts a character string as its argument, and returns a `casEventMask` object. The three main event types are provided as public members of `caServer`, so they don't need to be registered by calling `registerEvent()`. These three events are `valueEventMask`, `logEventMask`, and `alarmEventMask`, which are provided as public members of the `caServer` class. Currently, the client-side API only recognizes these three types of events.

Name:

`setDebugLevel()`

Synopsis:

```
#include <casdef.h>
void setDebugLevel (unsigned level);
```

Description:

The server tool can call `setDebugLevel()` to change the default debug level which is zero. The debug level determines how much run-time information is printed to the console during normal run-time operations and when `show()` is called. When the debug level is zero, the least amount of information--only error messages--are printed to the console. When the debug level is above two, the greatest amount of information is printed to the console.

Name:

`getDebugLevel()`

Synopsis:

```
#include <casdef.h>
unsigned getDebugLevel ();
```

Description:

The server tool can call `getDebugLevel()` to retrieve the level set by `setDebugLevel()`. Most likely, the server tool will use this function if it decides to provide an implementation of `caServer::show()`.

Name:

```
~caServer()
```

Synopsis:

```
#include <casdef.h>
virtual ~caServer();
```

Description:

`~caServer()` is the class destructor. Only in rare circumstances will the server tool want to call it explicitly. It is called automatically when any `caServer` or derived object goes out of scope, or, for objects on the free-store, when the object is deleted. According to the C++ standard, it is called after the destructor(s) of the derived class(es). The server tool doesn't have to know what it actually does, only that it will provide the appropriate cleanup for the server internals.

Name:

```
show()
```

Synopsis:

```
#include <casdef.h>
virtual void show (unsigned level);
```

Description:

`show()` is provided mostly for diagnostic purposes. It must be called by the server tool, unlike most virtual functions in the interface which are called by the server library. Its default implementation calls other `show()` functions from the server library's internal classes. As its only argument it accepts an unsigned integer representing the debug level, the level of diagnostic information printed out. Typically, the integer would be retrieved using `getDebugLevel()` and passed to `show()`.

You can redefine `show()` to print whatever. If you want to print a message before doing what the default version does, calling the `show()` function of internal classes, you can call the show function of the object pointed to by the `pCAS` private member.

Name:

`pvExistTest()`

Synopsis:

```
#include <casdef.h>
virtual pvExistReturn pvExistTest (const casCtx &ctx,
                        const char *pPVAliasName);
```

`pvExistTest()` is a virtual function. The two arguments to `pvExistTest()` are a `casCtx` object, `ctx`, and a character string, `pPVName`. The basic task of `pvExistTest()` is check to see if a PV identified by `pPVName` exists within the server. How the server tool determines if the PV exists is up to the server tool.

pvExistTest() should return `pvExistReturn` object whenever it's called. A `pvExistReturn` object is a container for a `pvExistReturnEnum` member. The `pvExistReturnEnum` type is an enumerated type whose values are `pverExistsHere`, `pverDoesNotExistHere`, and `pverAsyncCompletion`. The `pvExistReturn` object returned to the server library by `pvExistTest()` must have one of these values, which indicate that `pvExistTest()` found the PV in this server tool, didn't find the PV, or wishes to determine the existence of the PV asynchronously, respectively.

Although strictly speaking the return value of the `pvExistTest()` function is a `pvExistReturn` object, C++ allows a function to initialize and return an object indirectly to the calling function. Thus, `pvExistTest()` can simply return one of the three possible values of `pvExistReturnEnum`, in which case a `pvExistReturn` object will be returned to the server library initialized with the specified enumerator.

If the server tool returns `pverAsyncCompletion`, it should create a `casAsyncPVExistIO`, whose member function, `postIOCompletion()`, can be called to inform the server library that the server tool has completed the operation.

Name:

`createPV()`

Synopsis:

```
#include <casdef.h>
virtual pvCreateReturn createPV (const casCtx &ctx,
                                  const char *pPVAliasName);
```

Description:

The server library calls `createPV()` when a client is attached to the server tool for the first time. This occurs when a PV is found to exist in the server, i.e., when `pvExistTest()` returns `S_casApp_success`, and `createPV()` has not already been called for that PV before. In other words, when the first client requests a connection to a PV and the PV does in fact exist in the server, the server library will call `createPV()`.

The basic task of `createPV()` is to create an object of the `casPV` class or of a derived class and return a pointer to that object inside a `pvCreateReturn` object, which is a container object for a pointer to a `casPV` object as well as a status code. Because C++ allows a function to initialize and return an object indirectly to the calling function, `createPV()` can simply return a pointer to a PV object, a PV object (by reference), or a status code. The three possible status codes are `S_casApp_pvNotFound`, `S_casApp_noMemory`, and `S_casApp_asyncCompletion`. A `pvExistReturn` object will be created and returned to the server library, initialized with the appropriate members. If a NULL PV pointer is returned, the `pvCreateReturn` object is initialized with the status code `S_casApp_pvNotFound`.

If `createPV()` plans to return `S_casApp_asyncCompletion`, it should create a `casAsyncCreatePVIO` object, and call `postIOCompletion()` when the operation is completed.

`createPV()` accepts two arguments: a `casCtx` object, `ctx`, and a string, `pPVAliasName`. The argument `ctx` can be ignored except in cases of asynchronous completion, in which cases it should be passed to the constructor of the `casAsyncCreatePVIO` class. The argument `pPVAliasName` can be used to verify that the PV object exists, and it can be used to detect if the server library is attempting to create an object for a PV which already has had an object created for it.

Names:

```
valueEventMask
logEventMask
alarmEventMask
```

Synopsis:

```
#include <casdef.h>
const casEventMaskvalueEventMask;
const casEventMasklogEventMask;
const casEventMaskalarmEventMask;
```

Description:

These are the three main event types. They are currently the only ones recognized by clients. `registerEvent()` is called for these event types in the constructor to `caServer()`. `valueEventMask` is for value events, `logEventMask` is for archival events, and `alarmEventMask` is for alarm status events. Although it is possible to add events statically to the client-side API (before run-time). In the future, the client-side API will allow clients to register new types of events statically, as will the server-side API. For the server-side API, each event will be represented by a `casEventMask` object.

For now, the only `casEventMask` objects recognized by the server library are `valueEventMask`, `logEventMask`, and `alarmEventMask`. They can be passed to the `casPV::postEvent()` object. The `&` and `|` operators are overloaded so that they can be combined and separated.

CLASS: casPV

Declared: casdef.h

One `casPV` object must exist for each PV which a client has attached to. When `caServer::createPV()` is called, a `casPV` object should be created, though it's perfectly acceptable for a server tool to "precreate" a set of PV objects. In other words, it doesn't matter when a `casPV` or derived-class object is created, as long as one exists for a PV after `createPV()` is called. The server library calls `createPV()` when the first client attaches to a PV.

The `casPV` class itself consists of virtual and non-virtual functions that together satisfy client requests to read, write, and monitor channels. The `casPV` class has one pure virtual function, `getName()`, so it is an abstract class. A server tool must derive a class from `casPV` and redefine `getName()` so that it returns a string representing the class name. As with the other base classes in the server interface, the default implementations of the `casPV` virtual functions are empty or trivial implementations so that a server tool can create a `casPV` object without having to provide implementations for virtual functions which it doesn't intend to use. All virtual functions in the `casPV` class are called by the server library.

The most important virtual functions which the server tool will probably want to provide definitions for are the `read()` and `write()` functions which satisfy a client's request to read a PV's value (and possibly its attributes) and to change a PV's value.

Destruction:

The `destroy()` virtual function is provided as part of the `casPV` class to aid the server tool in destroying all `casPV` objects. The server library calls `destroy()` for a PV either when the last client disconnects from a PV or when the server tool itself is deleted, i.e., when `caServer` is deleted or destroyed, in which case the `destroy()` member function of each `casPV` object is called. The default version of `destroy()` does a delete this. Since the `delete` operator only works for objects on the free store, unless the `casPV` object was created with the new operator in `createPV()`, the default version of `destroy()` won't work. Therefore, if the server tool creates `casPV` objects by another method other than the `new` operator, it should redefine the `destroy()` function in a derived class.

The server interface also provides a virtual destructor, `~casPV()`. This destructor will provide the necessary cleanup for the internals of the `casPV` class before an object's container is destroyed. If the server tool derives a class from the `casPV` class, it is responsible for cleaning up any members that need to be explicitly deleted (such as members on the free-store) before the object's container is destroyed. It should do this in its own destructor. Recall that when the destructor of a base class is declared `virtual`, the destructors of all classes derived from that class will be `virtual`. Virtual destructors differ from other virtual functions in that the base class' destructor is not really a default, but instead guarantees a certain order of execution of destructors in the derivation hierarchy before an object is destroyed.

For example, if a server tool derives a class from `casPV` called `aPV` and in `createPV()` creates `aPV` objects using the `new` operator, the default version of `destroy()` will correctly cause destroy all `aPV` objects. If `aPV` provides its own destructor, then when the server library calls `destroy()`, the `delete` operator will cause the object to be destroyed. Before the object's container is actually destroyed, the destructors of the `casPV` derivation hierarchy will be executed, that is, the derived class destructor will execute, `~aPV()`, and then the base class destructor will execute, `~casPV()`.

Public Member Functions:

Non-Virtual

```
casPV (const casCtx &ctx, const char * const pPVName);
caServer *getCAS();
void postEvent (const casEventMask &select, gdd &event);
```

Virtual

```
virtual ~casPV ();
virtual void show (unsigned level);
virtual caStatus interestRegister ();
virtual void interestDelete ();
virtual caStatus beginTransaction ();
virtual void endTransaction ();
virtual caStatus read (const casCtx &ctx, gdd &prototype);
virtual caStatus write (const casCtx &ctx, gdd &value);
virtual casChannel *createChannel(const casCtx &ctx,
                      const char * const pUserName,
                      const char * const pHostName);
virtual void destroy ();
virtual aitEnum bestExternalType ();
virtual unsigned maxDimension() const;
virtual aitIndex maxBound (unsigned dimension) const;
```

Pure Virtual

```
virtual const char *getName() const = 0;
```

Name:

`casPV()`

Synopsis:

```
#include <casdef.h>
casPV (caServer &cas);
```

Description:

`casPV()` is the class constructor. The server tool shouldn't be concerned with what the constructor does, only with passing it the correct argument, the `caServer` object (or derived class) that is associated with the current PV.

Name:

`getCAS()`

Synopsis:

```
#include <casdef.h>
caServer *getCAS();
```

Description:

`getCAS()` returns a pointer to the `caServer` object associated with the current PV. The server tool may need to access the PV's `caServer` object and its member functions for any one of several purposes. However, this function is used mostly by the server library.

Name:

`postEvent()`

Synopsis:

```
#include <casdef.h>
void postEvent (const casEventMask &select, gdd &event);
```

Description:

The server tool calls `postEvent()` to post events for the PV. Its two arguments are a `casEventMask` object, `select`, and a `gdd` object, `event`. `select` can be any event mask currently registered with the server library. In addition, `select` can be any combination of event masks. This can be done by ORing (`valueEventMask | logEventMask`) any combination of the masks together. `event` should be the value to be posted, such as the new value or alarm status.

Name:

`~casPV()`

Synopsis:

```
#include <casdef.h>
virtual ~casPV ();
```

Description:

`~casPV()` is the `casPV` class destructor. Only in freak circumstances will the server tool call it directly. It is called automatically when any `casPV` or derived class object goes out of scope, or, for objects on the free-store, when the object is deleted. According to the C++ standard, it is called after the destructor(s) of any derived class(es). The server tool doesn't have to know what it actually cleans, only that it will provide the appropriate cleanup for the `casPV` internals.

Name:

`show()`

Synopsis:

```
#include <casdef.h>
virtual void show (unsigned level);
```

Description:

`show()` is called if `caServer::show()` is called and the debug level is high enough. The default definition of `casPV::show()` prints out the best external type, i.e., the type returned by `bestExternalType()`. The server tool can redefine `show()` as it sees fit.

Names:

`interestRegister(), interestDelete()`

Synopsis:

```
#include <casdef.h>
virtual caStatus interestRegister ();
virtual void interestDelete ();
```

Description:

The server library calls `interestRegister()` when the first client establishes a monitor on the PV, and it calls `interestDelete()` when the last client removes its monitor on the PV. Thus, these functions are meant to indicate whether or not a client or clients are monitoring the PV.

The default versions of these functions are empty or NULL functions, so if the server tool wishes to implement monitors, it must redefine both of these functions. The basic task of these functions is to provide a way to indicate whether or not any clients have monitors on a PV. One way to do this is for `interestRegister()` to set a member True and `interestDelete()` to set it False. `interestRegister()` must return a status code to the server library.

Names:

`beginTransaction(), endTransaction()`

Synopsis:

```
#include <casdef.h>
virtual caStatus beginTransaction ();
virtual void endTransaction ();
```

Description:

The server library calls `beginTransaction()` before it calls either `read()` or `write()`. It calls `endTransaction()` after a call to either `read()` or `write()`. The default implementation of `beginTransaction()` merely returns `S_casApp_success`, and the default version of `endTransaction()` is empty. The server tool can redefine them in a derived class to perform any pre-IO or post-IO tasks which it needs. If `beginTransaction()` returns an error code as opposed to `S_casApp_success`, the server library will not call `read()` or `write()`.

Name:

`read()`

Synopsis:

```
#include <casdef.h>
virtual caStatus read (const casCtx &ctx, gdd &prototype);
```

Description:

The server library calls `read()` when a client requests a read operation for the PV and `beginTransaction()` returns `S_casApp_success`. Its two arguments are a `casCtx` object, `ctx`, and a `gdd` object, prototype. `ctx` is used by the server library when asynchronous IO is involved; the server tool should not be concerned with it, except that for asynchronous operations it should be passed to the constructor for the `casAsyncReadIO` class. `prototype` can be a `gddScalar`, a `gddAtomic`, or a `gddContainer` object. The basic task of `read()` is to write the PV's value into the `gdd` object. In addition, if prototype is a `gddContainer` object, `read()` should write the PV's value as well as its attributes into the `gdd` container object.

Each `gdd` object passed to `read()` should have an application type, returned by `gdd::applicationType()`. An application type refers to an unsigned integer code that represents a predefined use for a piece of data. For example, one application type is the constant `gddAppType_status` which identifies the value contained in a `gdd` object as representing an alarm status code. New application types can be defined to encompass new applications. Application types for EPICS applications can be found in gddApps.h.

If prototype is a `gddScalar` or `gddAtomic`, then its application type is most likely `gddAppType_value`, at least for EPICS applications. When the object is `gddAppType_value`, then `read()` must write the PV's value into the object using `gdd::put()` or `gdd::putConvert()`. If prototype is a `gddScalar` object, then a single value must be written into it. If a `gddAtomic` object, then the appropriate number of elements must be written into `prototype`. If `prototype` is a `gddContainer`, then that means that it itself contains other `gdd` objects, each having its own application type. `read()` must then provide a way to step through the container and its objects, writing the appropriate value that corresponds to the objects' application type into each object.

There are several ways to step through a `gdd` container and call the appropriate functions. The easiest way is to use the `gddAppFuncTable<PV>` template. This class provides a way to install functions and to specify an application type for each function. Then when `gddAppFuncTable::read()` is called and provided with the appropriate arguments, it will step through the `gdd` container object, calling the appropriate function for each `gdd` object.

Basically, the `read()` function must write the appropriate value or values into `prototype` in order to satisfy client requests. Whatever method it chooses to implement this functionality will work as long as it provides a way to satisfy any expected client requests. `read()` must return a status code to the server library. The appropriate code and a message will be sent to the client.

Normally, `read()` completes synchronously. However, `read()` can complete asynchronously creating a `casAsyncReadIO` or derived object and returning the status code `S_casApp_asyncCompletion`.

Name:

`write()`

Synopsis:

```
#include <casdef.h>
virtual caStatus write (const casCtx &ctx, gdd &value);
```

Description:

The server library calls `write()` when a client makes a "put" or write request for the PV and after `beginTransaction()` returns `S_casApp_success`. Its two arguments are a `casCtx` object, `ctx`, and a `gdd` object, `value`. `ctx` is used by the server library when asynchronous IO is involved; the server tool should not be concerned with it, except that for asynchronous operations it should be passed to the constructor for the `casAsyncWriteIO` class. `value` contains the value which the client has requested the server tool to write to the PV. The server tool must somehow update the current value of the PV. The ways to do this are numerous and depend totally on the server tool. The default version of `write()` is a NULL function.

`write()` is somewhat easier to implement than `read()` because as of release 3.13, the client-side API doesn't allow put or write requests for compound types. Thus, `value` will never be a `gddContainer` object, but either a `gddAtomic` or `gddScalar` object. The task of `write()` will be, for `gddAtomic` objects, to write the elements from the array to the PV's array and, for `gddScalar` objects, to write the `gddScalar` value to the PV's value. `write()` should never have to step through a `gddContainer` object. `write()` should return `S_casApp_success` if the operation was successful, or the appropriate error code if the operation was not.

Like `read()`, `write()` can complete asynchronously by creating a `casAsyncIO` or derived-class object and returning the status code `S_casApp_asyncCompletion` to the server library.

Name:

```
createChannel()
```

Synopsis:

```
#include <casdef.h>
virtual casChannel *createChannel(const casCtx &ctx,
                       const char * const pUserName,
                       const char * const pHostName);
```

Description:

Recall that the term channel refers to the connection between a PV and a client. The server library calls `createChannel()` for each new client that attaches to the PV. The default version of `createChannel()` simply creates a `casChannel` object, passing the first argument, `ctx`, to its constructor.

A server tool will only want to redefine `createChannel()` when it wishes to implement access control. To restrict access rights, it will have to derive a new class from the `casChannel` class and redefine the proper functions. `casChannel()` can be redefined to create an object of the derived class and can use the second and third arguments, `pUserName` and `pHostName` to grant or restrict a a client access to read and/or write to a PV. `pUserName` points to a string of the user's name, while `pHostName` points to a string of the host's name. If the server tool is going to restrict access rights, it will probably use these two names.

`createChannel()` should return a pointer to the `casChannel` or derived-class object to the server library.

Name:

`destroy()`

Synopsis:

```
#include <casdef.h>
virtual void destroy ();
```

Description:

The server library calls destroy when the last client disconnects from the PV or when the server itself (the `caServer` object) is deleted, in which case the `destroy()` functions of all `casPV` or derived-class objects are called. The default version of `destroy()` deletes the `this` pointer, so unless the `casPV` or derived-class object is on the free store, the server tool will have to redefine `destroy()` to delete the `casPV` or derived-class objects.

Name:

`bestExternalType()`

Synopsis:

```
#include <casdef.h>
virtual aitEnum bestExternalType ();
```

Description:

`bestExternalType()` returns the recommended architecture-independent type for requests, one of the enumerators of the enumerated type `aitEnum`. The server tool can recommend a type because it's most efficient, or because it's the easiest, or because it matches the PV's type. The default version of `bestExternalType()` returns `aitEnumString`.

The server library calls `bestExternalType()` for its own purposes as well as to satisfy client requests for the PV's native type.

Name:

`maxDimension(), maxBound()`

Synopsis:

```
#include <casdef.h>
virtual unsigned maxDimension() const;
virtual aitIndex maxBound (unsigned dimension) const;
```

Description:

`maxDimension()` and `maxBound()` are virtual functions that a server tool must redefine to accurately describe the "size" of the PV's value. For scalar PVs, both functions should return zero. The default versions of these functions both return zero, so by default they describe a scalar PV. Thus, a server tool will want to redefine these functions for array PVs.

For an array PV, `maxDimension()` should return an integer representing the number of dimensions of the PV's array. For a one dimensional array, it should return one. For a two dimensional array, it should return two, and so on. `maxBound()` should return the size of the dimension indexed by the integer passed to it. If passed an argument of zero, for instance, it should return the number of elements in the first dimension of the array. If passed an argument of one, it should return the number of elements in the second dimension and so on. Note that for scalar PVs, `maxBound()` should return zero no matter what dimension its argument specifies.

CLASS: casChannel

Declared: casdef.h

A `casChannel` object or an object of a derived class must exist for each client connection to each PV. Thus, if client X and client Y have each established connections to PVs A and B, then there are four channels: two channels on PV A, one for client X and one for Y; and two channels on PV B, one for client X and one for Y. Each time a new client attaches to a PV, the server library calls `casPV::createChannel()`. `createChannel()` is a virtual function whose default creates a `casChannel` object.

The `casChannel` class provides the server tool with the ability to restrict or control access rights to PV. It consists of some virtual and non-virtual functions. By deriving a class from the `casChannel` class and redefining the appropriate virtual functions, a server tool can control access rights to a PV. By default, none of the `casChannel` functions implement access control. By default, most are empty or NULL functions.

Access rights consist of read access and write access. The server tool can grant one and deny the other, or grant both, or deny both, whatever it deems to be appropriate.

Basically, to implement access rights, a server tool needs to derive a class from the `casChannel` class. Recall that the server library calls `casPV::createChannel()` each time a client attaches to a PV. Two of the arguments passed to `createChannel()` are the user's name and the client's host name. If the server tool wishes to control access rights based upon who the user is and on what machine the client is running, it should provide a way to keep track of the user and host names, such as providing members to keep track of the names as part of a class derived from `casChannel`.

The two basic functions that control access rights are `readAccess()` and `writeAccess()`. Both of these return an enumerated type, `aitBool`, whose enumerators are `aitTrue` and `aitFalse`. If `readAccess()` returns `aitFalse`, the client will be denied read access to the PV, i.e., will not be able to read its value. If `readAccess()` returns `aitTrue`, the client will be granted read access to the PV. `writeAccess()` works the same, except that whether it returns `aitTrue` or `aitFalse` controls whether or not a client has write access to a PV, i.e., can change its value.

Both `readAccess()` and `writeAccess()` are virtual functions whose default definitions always return `aitTrue`. A server tool can derive a class from `casChannel` and then redefine these functions to grant/deny access depending on whatever condition is deems appropriateÄfor example, because the user's name is 'hacker' or because write operations are not allowed after 5:00 PM.

Destruction:

The server tool should not itself directly delete or destroy a `casChannel` or derived-class object. Instead, the server library calls the virtual function `destroy()`. It calls `destroy()` in any one of the following three situations:

- When a client disconnects from a PV, the server library deletes the `casChannel` or derived-class object for the client.
- When a PV is destroyed, the server library deletes all `casChannel` or derived-class objects associated with the PV.
- When a server is destroyed, all PVs are deleted, before which all `casChannel` or derived-class objects associated with all PVs are destroyed.

The default version of `destroy()` deletes the `this` operator using the `delete` operator. Thus, it will work only if the object is on the free store, i.e., was created using the `new` operator. If the object is not on the free store, `destroy()` will have to be redefined so that it duly deletes or destroys the derived-class object.

The `casChannel` class has a virtual destructor, `~casChannel()`. The destructor cleans up any internals related to the `casChannel` class and its internals. If the server tool derives a class from the `casChannel` class, it is responsible for providing a constructor that will perform the necessary cleanup. Remember that a virtual destructor in the base class automatically makes all destructors in the derived classes virtual, whether declared so or not. Virtual destructors are always called in the same order when an object is deleted: the derived class constructor first, then the constructor of next class(es) in the hierarchy, and so on until the destructor(s) of the base class(es) are called. After all destructors are called, the object's container is destroyed.

Public Member Functions:

Non-Virtual

```
casChannel(const casCtx &ctx);
casPV *getPV();
void postAccessRightsEvent ();
```

Virtual

```
virtual void destroy();
virtual void show(unsigned level);
virtual aitBool confirmationRequested () const;
virtual aitBool readAccess () const;
virtual aitBool writeAccess () const;
virtual void setOwner(const char * const pUserName,
                   const char * const pHostName);
```

Name:

`casChannel()`

Synopsis:

```
#include <casdef.h>
casChannel(const casCtx &ctx);
```

Description:

`casChannel()` is the class constructor. It initializes the internals of `casChannel` and its base class. The server tool should not be concerned with what it does, but should only make sure that the proper `casCtx` object gets passed as its argument.

Name:

`getPV()`

Synopsis:

```
#include <casdef.h>
casPV *getPV();
```

Description:

`getPV()` returns a pointer to the `casPV` or derived-class object associated with the current `casChannel` or derived-class object.

Name:

`readAccess()`

Synopsis:

```
#include <casdef.h>
virtual aitBool readAccess () const;
```

Description:

The server library calls `readAccess()` to determine if a client should be granted access to read a PV's value. `readAccess()` returns an enumerated type, `aitBool`, whose enumerators are `aitTrue` or `aitFalse`. If `readAccess()` returns `aitTrue`, the client will be granted access to read the PV. If it returns `aitFalse`, the client will be denied access to read the PV.

The default version of `readAccess()` always returns `aitTrue`. A server tool can control access to a PV by returning `aitTrue` or `aitFalse` as it deems appropriate, based on whatever conditions are relevant, though usually it will deny/grant access according to the user name and host name passed to `createChannel()`.

It's entirely valid for `readAccess()` to return different values throughout a channel's lifetime, that is, for `readAccess()` to return `aitFalse` at one time, and then return `aitTrue` at another time. The client will be denied/granted access accordingly.

Name:

`writeAccess()`

Synopsis:

```
#include <casdef.h>
virtual aitBool writeAccess () const;
```

Description:

The server library calls `writeAccess()` to determine if a client should have write access to a PV. `writeAccess()` returns the enumerated type `aitBool`, whose enumerators are `aitFalse` and `aitTrue`. If `writeAccess()` returns `aitTrue`, the client will be granted access to write to the PV. If it returns `aitFalse`, the client will be denied access to write to the PV.

The default version of `writeAccess()` always returns `aitTrue`. A server tool can control write access to a PV by redefining the function to return `aitFalse` or `aitTrue` as it deems appropriate, based on whatever conditions it chooses to use, though usually it will deny/grant access according to the user name and host name passed as arguments to `casPV::createChannel()`.

It's entirely valid for `writeAccess()` to return different values throughout a channel's lifetime, that is, for `writeAccess()` to return `aitFalse` at one time, and then return `aitTrue` at another time. The client will be denied/granted access accordingly.

Name:

`show()`

Synopsis:

```
#include <casdef.h>
virtual void show(unsigned level);
```

Description:

If the default version of `caServer::show()` is called and the level is high enough, the server library will call `show()`. If the debug level is above two, the default version of `show()` will print the values returned by `readAccess()`, `writeAccess()`, and `confirmationRequested()`. The server tool can redefine `show()` to print whatever information it deems appropriate.

Name:

`setOwner()`

Synopsis:

```
#include <casdef.h>
virtual void setOwner(const char * const pUserName,
                   const char * const pHostName);
```

Description:

If either the user name or host name should change throughout the life of the channel, the server library calls `setOwner()` and passes the names of the user and host to the function. The default version of `setOwner()` is a NULL or empty function. If the server tool wishes to control access rights based upon who the user is and the machine the client is running on, it should redefine `setOwner()` so that it can keep track of the new names.

Name:

```
confirmationRequested()
```

Synopsis:

```
#include <casdef.h>
virtual aitBool confirmationRequested () const;
```

Description:

The server library calls `confirmationRequested()` to determine if the OPI should prompt the user for confirmation before writing to this PV. `confirmationRequested()` returns an enumerated type, `aitBool`, whose enumerators are `aitTrue` and `aitFalse`. If it returns `aitTrue`, the OPI display will prompt the user for confirmation when the user tries to write to the PV. If it returns `aitFalse`, no such prompting will occur when the user writes to a PV.

Currently, neither DM nor the client-side API allows for such a confirmation request. This is meant to change in a future EPICS release.

Name:

```
destroy()
```

Synopsis:

```
#include <casdef.h>
virtual void destroy();
```

Description:

The server library calls `destroy()` to delete the `casChannel` or derived-class object. It calls destroy when a client disconnects from a PV, when a PV is deletedÄin which case `destroy()` is called for all `casChannel` or derived-class objects associated with the PVÄor when a server is deleted, all PVs associated with the server are deleted, in which case `destroy()` is called for all `casChannel` or derived-class objects associated with the PV.

The default version of `destroy()` deletes the object's `this` pointer. Since it uses the `delete` operator, it is only valid for objects created using the `new` operator, objects on the free store. If the server tool creates `casChannel` or derived-class objects using another method other than the `new` operator, it should redefine the `destroy()` function so that the object is duly deleted when the server library calls `destroy()`.

Name:

`postAccessRightsEvent()`

Synopsis:

```
#include <casdef.h>
void postAccessRightsEvent ();
```

Description:

The Channel Access client-side API allows a client to monitor a PV for access rights events. That is, a client can request the server tool to inform the client when the client's access rights change, when it doesn't have access rights where it did before or when it has access rights where it didn't before.

If a server tool implements access control and the access for a particular channel changes during the channel's lifetime, then `postAccessRightsEvent()` should be called for the `casChannel` object associated with the channel. The server library will then inform the client of the access rights event.

CLASS: casAsyncIO

Declared: casdef.h

The casAsyncIO class is provided as a base class to the casAsyncReadIO, casAsyncWriteIO, casAsyncCreatePVIO, and casAsyncPVExistIO classes. These classes allow asynchronous completion of casPV::read(), casPV::write(), caServer::createPV(), and caServer::pvExistTest(), respectively. If one of these functions chooses to complete asynchronously, it must create a casAsyncReadIO, casAsyncWriteIO, casAsyncCreatePVIO, or casPVExistIO object, and then use the postIOCompletion() function to post the necessary status codes and values. postIOCompletion() is a member of all four of the above classes.

The `casAsyncIO` class provides for a consistent method of destruction for the `casAsyncReadIO`, `casAsyncWriteIO`, `casAsyncCreatePVIO`, and `casAsyncPVExistIO` classes. It provides a virtual destructor `~casAsyncIO()` and a virtual function called `destroy()`. The server library calls `destroy()` after the message posted using `postIOCompletion()` has been successfully queued to the client. The default version of `destroy()` deletes the `this` pointer of the object using the `delete` operator. Thus, if the object was created using the `new` operator and is on the free store, the default version of `destroy()` will cause the object's destructor to be called and the object's container to be destroyed. If the server tool wishes to create a `casAsyncReadIO`, `casAsyncWriteIO`, `casAsyncCreatePVIO`, or `casAsyncPVExistIO` object by another method than the `new` operator, it should redefine the `destroy()` function in a derived class if it wants the object to be duly destroyed.

The virtual destructor `~casAsyncIO()` performs no real clean up, but instead guarantees the order in which the destructors of any derived classes are called. Since the `casAsyncReadIO`, `casAsyncWriteIO`, `casAsyncCreatePVIO`, and `casAsyncPVExistIO` classes are derived from `casAsyncIO`, if the server tool derives a class from any of these classes, when the object is destroyed, the destructor of the derived class is called first, the destructor of the `casAsyncReadIO`, `casAsyncWriteIO`, `casAsyncCreatePVIO`, or `casAsyncPVExistIO` class is called second, and finally `~casAsyncIO()` is called.

Public Member Functions:

Virtual

```
virtual ~casAsyncIO();
virtual void destroy();
```

Name:

`~casAsyncIO()`

Synopsis:

```
#include <casdef.h>
virtual ~casAsyncIO();
```

Description:

`~casAsyncIO()` is the class destructor. It is an empty or NULL function, and is only provided to guarantee the execution order of the destructors of the `casAsyncReadIO`, `casAsyncWriteIO`, `casAsyncCreatePVIO`, and `casAsyncPVExistIO` class, as well as any classes derived from these classes.

The server tool should never call this destructor directly. The server tool should also never directly delete any asynchronous IO object such as `casAsyncReadIO`.

Name:

`destroy()`

Synopsis:

```
#include <casdef.h>
virtual void destroy();
```

Description:

The server library calls `destroy()` after the message posted by `postIOCompletion()` has been successfully queued to the client. The default version of `destroy()` deletes the `this` pointer to the object using the `delete` operator. Since the `delete` operator only deletes objects on the free store, i.e., those created using the `new` operator, the default version will only work if the `casAsyncReadIO`, `casAsyncWriteIO`, `casAsyncCreatePVIO`, `casAsyncPVExistIO`, or derived-class object was created using the `new` operator. If the asynchronous object is to be created without the `new` operator, `destroy()` should be redefined so that it duly destroys the object.

Note that the server library calls `destroy()`. The server tool should never directly call `destroy()`. This may cause the message to be deleted before the server library has had a chance to queue it to the client. If the server tool wishes to cancel an asynchronous operation after the asynchronous object has already been created, it should call `postIOCompletion()` and pass it the status code `S_casApp_canceledAsyncIO`. `PostIOCompletion()` is a member function of all four asynchronous classes.

CLASS: casAsyncReadIO

Declared: casdef.h

The `casAsyncReadIO` class provides asynchronous completion for read operations. A read operation is one in which the client requests a value of a PV from the server. When such an operation is requested, the server library calls `casPV::read()` or a redefinition of it in a derived class. The server tool can choose to perform the operation in `read()` or to postpone completing the operation, called asynchronous completion, by creating a `casAsyncReadIO` or derived object and then returning `S_casApp_asyncCompletion`.

How the server tool chooses to complete an asynchronous operation depends on the application and is the concern of the server tool. For instance, the server tool could use a callback function, or a timer, or whatever. After the server tool completes the read operation, it should call `postIOCompletion()`, which is a member of the class. If the operation is successfully completed, the status code `S_casApp_success` should be passed as the first argument to `postIOCompletion()`, and a `gdd` object that contains the requested value should be passed as the second argument. The server library will create a message containing the value and the status code and put this message on a queue to be sent to the client. If the operation is unsuccessful, the appropriate error code should be passed as the first argument. A `gdd` object must be passed as the second argument, but this object will be ignored if the status code is not `S_casApp_success`.

If the server tool wishes to cancel an asynchronous IO operation, it should call `postIOCompletion()` and pass to it the status code `S_casApp_canceledAsyncIO`. The server library will then take care of destroying the object. The server tool should never delete an asynchronous object itself. Deleting the object before the server library is able to queue it to the client will cause an error.

Destruction:

When the server library has successfully queued the response message posted by `postIOCompletion()`, when the server tool cancels the IO operations, or when the client disconnects before the asynchronous operation completes, the server library will call the `destroy()` virtual function. `destroy()` is a member of the `casAsyncIO` class, a class from which `casAsyncReadIO` is derived. It deletes the `this` pointer, therefore causing the object to be destroyed if the object is on the free store. If the object was not created with `new`, `destroy()` should be redefined in a derived class so that the object is duly destroyed. The server tool should never directly call `destroy()`, should never directly call a `casAsyncReadIO` or derived-class destructor, and should never itself `delete` a `casAsyncReadIO` object. This is because a `casAsyncReadIO` object should not be destroyed until the server library has successfully queued the response message.

The `casAsyncIO` and `casAsyncReadIO` classes each have a virtual destructor, `~casAsyncIO()` and `~casAsyncReadIO()`. Because the destructors are virtual, they guarantee that the destructors in the `casAsyncReadIO` class and any derived classes are executed in order. If the server tool derives a class from `casAsyncReadIO`, it is responsible for providing a destructor that will perform any necessary clean up for that class prior to destruction.

For example, suppose a server tool derived a class called `myAsyncReadIO` from `casAsyncReadIO` and provided a destructor for the class, `~myAsyncReadIO()`. If `a myAsyncReadIO` object is created, when the operation completes and the server tool calls `postIOCompletion()`, the server library will queue the response to the client, after which it will call `destroy()`. Assuming that the `myAsyncReadIO` object was created using the `new` operator, then the default version of `destroy()` will delete the object, causing `~myAsyncReadIO()` to be called before `~casAsyncReadIO()` is executed, after which `~casAsyncIO()` is called, after which the object's container is destroyed.

Public Member Functions:

Non-Virtual

```
casAsyncReadIO(const casCtx &ctx);
caStatus postIOCompletion(caStatus completionStatusIn,
                   gdd &valueRead);
caServer *getCAS();
```

Virtual

```
virtual ~casAsyncReadIO();
```

Name:

`casAsyncReadIO()`

Synopsis:

```
#include <casdef.h>
casAsyncReadIO(const casCtx &ctx);
```

Description:

casAsyncReadIO() is the class constructor. As with the other constructors, the server tool doesn't need to be concerned with what it does, only in passing it the right arguments. Its only argument is a casCtx object, ctx. ctx is used by the server library; the server tool need not be concerned with it, except as far as making sure that it gets passed to the constructor.

Name:

`postIOCompletion()`

Synopsis:

```
#include <casdef.h>
caStatus postIOCompletion(caStatus completionStatusIn,
                   gdd &valueRead);
```

Description:

The server tool should call `postIOCompletion()` after it has completed the read operation. If successful, the first argument should be the status code `S_casApp_success` and the second argument should be a `gdd` object that contains the requested value (or values when the client has requested a compound type). If the read operation was not successful, the first argument should be an error code and the second argument a `gdd` object whose contents are not important.

If the server tool wishes to cancel an asynchronous read operation, it should call `postIOCompletion()` with the first argument the status code `S_casApp_canceledAsyncIO` and the second argument an empty `gdd` object. The server library will then call `destroy()` and forward the appropriate status code to the client.

Name:

```
getCAS()
```

Synopsis:

```
#include <casdef.h>
caServer *getCAS();
```

Description:

`getCAS()` returns a pointer to the `caServer` object associated with the `casAsyncReadIO` or derived-class object.

Name:

```
~casAsyncReadIO()
```

Synopsis:

```
#include <casdef.h>
virtual ~casAsyncReadIO();
```

Description:

`~casAsyncReadIO()` is the class destructor. It is an empty or NULL function, and is only provided to guarantee the execution order of any destructors declared in any derived class or classes. Thus, if a server tool derives a class from the `casAsyncReadIO` class and provides a destructor for that class, when the object is deleted or destroyed, the destructor for the derived class will be called before the destructor of the base class.

The server tool should never call this destructor directly. The server tool should also never directly delete a `casAsyncReadIO` object.

CLASS: casAsyncWriteIO

Declared: casdef.h

The `casAsyncWriteIO` class provides asynchronous completion for write operations. A write operation is one in which the client requests that a PV's value be changed. The client, of course, provides the value. When such an operation is requested, the server library calls `casPV::write()` or a redefinition of it in a derived class. The server tool can choose to perform the operation in `write()`, synchronous completion, or to postpone completing the operation, asynchronous completion, in which case `write()` should create a `casAsyncWriteIO` object and then return `S_casApp_asyncCompletion`.

How the server tool chooses to complete the operation depends on the application and is the concern of the server tool. For instance, the server tool could use a callback function, or a timer, or whatever. After the server tool completes the write operation, it should call `postIOCompletion()`. If the operation is successfully completed, the status code `S_casApp_success` should be passed to `postIOCompletion()`. The server library will create a message consisting of the status and put this message on a queue to be sent to the client. No `gdd` object should be posted for asynchronous write operations.

The value to be written to the PV in the operation is contained in the `gdd` object that the server library sends to `write()`. The server tool will need to keep track of this `gdd` object or the value inside it, and then write this value to the PV when the server tool completes the operation. The easiest way to do this would be to derive a class from the `casAsyncWriteIO` class. The derived class would have a member that would be assigned the gdd object passed to `write()`. When the operation is ready to complete, this member can be accessed and its value written to the PV.

If the server tool wishes to cancel an asynchronous IO operation, it should call `postIOCompletion()` and pass it the status code `S_casApp_canceledAsyncIO`. Deleting the object before the server library is able to queue it to the client will cause error.

Destruction:

When the server library has successfully queued the response message posted by `postIOCompletion()` or when the client disconnects before the asynchronous operation completes, the server library will call the `destroy()` virtual function. `destroy()` is a member of the `casAsyncIO` class, a class from which `casAsyncWriteIO` is derived. `destroy()` deletes the `this` pointer to the object, therefore causing the object to be destroyed if the object is on the free store. If the object is not created with `new`, `destroy()` should be redefined in a derived class so that the object is duly destroyed. The server tool should never directly call `destroy()`, should never directly call a `casAsyncIO` or derived-class destructor, and should never itself delete a `casAsyncWriteIO` object. This is because a `casAsyncWriteIO` object should not be destroyed until the server library has successfully queued the response message to the client.

The `casAsyncIO` and casAsyncWriteIO classes each have a virtual destructor, `~casAsyncIO()` and `~casAsyncWriteIO()`. Because the destructors are virtual, they guarantee that the destructors in the `casAsyncWriteIO` class and any derived classes are executed in order. If the server tool derives a class from `casAsyncWriteIO`, it is responsible for providing a destructor that will perform any necessary clean up prior to destruction.

For example, suppose a server tool derived a class from `casAsyncWriteIO` called `myAsyncWriteIO` and provided a destructor for the class, `~myAsyncWriteIO()`. If `a myAsyncWriteIO` object is created, when the operation completes and the server tool calls `postIOCompletion()`, the server library will queue the response to the client, after which it will call `destroy()`. Assuming that the `myAsyncWriteIO` object was created using the `new` operator, then the default version of `destroy()` will delete the object causing `~myAsyncWriteIO()` to be executed before `~casAsyncWriteIO()` is executed, after which `~casAsyncIO()` is called, after which the object's container is destroyed.

Public Member Functions:

Non-Virtual

```
casAsyncWriteIO(const casCtx &ctx);
caStatus postIOCompletion(caStatus completionStatusIn);
caServer *getCAS();
```

Virtual

```
virtual ~casAsyncWriteIO();
```

Name:

`casAsyncWriteIO()`

Synopsis:

```
#include <casdef.h>
casAsyncWriteIO(const casCtx &ctx);
```

Description:

`casAsyncWriteIO()` is the class constructor. As with the other constructors, the server tool doesn't need to be concerned with what it does, only in passing it the right arguments. Its only argument is a `casCtx` object, `ctx`. `ctx` is used by the server library; the server tool need not be concerned with it, except as far as making sure that it gets passed to the constructor.

Name:

`postIOCompletion()`

Synopsis:

```
#include <casdef.h>
caStatus postIOCompletion(caStatus completionStatusIn);
```

Description:

The server tool should call `postIOCompletion()` after it has completed the write operation. If successful, the status should be `S_casApp_success`. If the write operation was not successful, the status should be the appropriate error code.

If the server tool wishes to cancel an asynchronous read operation, it should call `postIOCompletion()` with the status code `S_casApp_canceledAsyncIO`. The server library will then call destroy and forward the appropriate status code to the client.

Name:

`getCAS()`

Synopsis:

```
#include <casdef.h>
caServer *getCAS();
```

Description:

`getCAS()` returns a pointer to the `caServer` object associated with the `casAsyncWriteIO` or derived-class object.

Name:

`~casAsyncWriteIO()`

Synopsis:

```
#include <casdef.h>
virtual ~casAsyncWriteIO();
```

Description:

`~casAsyncWriteIO()` is the class destructor. It is an empty or NULL function, and is only provided to guarantee the execution order of any destructors declared in any derived class or classes. Thus, if a server tool derives a class from the `casAsyncWriteIO` class and provides a destructor for that class, when the object is deleted or destroyed, the destructor for the derived class will be called before the destructor of the base class.

The server tool should never call this destructor directly. The server tool should also never directly delete a `casAsyncWriteIO` object.

CLASS: casAsyncPVExistIO

Declared: casdef.h

The `casAsyncPVExistIO` class provides asynchronous completion for `pvExistTest()`. The server library calls `pvExistTest()` when a client requests a connection to a PV, broadcasting the PV's name over the network. The server library will pick up the broadcast and call the server tool's implementation of `pvExistTest()`. The task of `pvExistTest()` is to check to see if the PV is associated with this server, and if it is, to return `pverExistsHere`. If it isn't, it should return `pverDoesNotExistHere`. If the server tool wants to perform the search later, it can postpone the operation using asynchronous completion, in which case it should create a `casAsyncPVExistIO` object and then return `pverAsyncCompletion`. Of course, like all asynchronous objects, the `casAsyncPVExistIO` object must exist after the function which created it returns.

Although strictly speaking the return value of the `pvExistTest()` function is a `pvExistReturn` object, C++ allows a function to initialize and return an object indirectly to the calling function. Thus, `pvExistTest()` can simply return one of the three possible values of `pvExistReturnEnum`, in which case a `pvExistReturn` object will be returned to the server library initialized with the specified enumerator. A `pvExistReturn` object is a container for a `pvExistReturnEnum` member. For asynchronous completion, a `pvExistReturn` object must be explicitly initialized and passed to `postIOCompletion()`. A `pvExistReturn` object can be initialized simply by passing it a `pvExistReturnEnum` value:

```
pvExistReturn *pPver = new pvExistReturn(pverExistsHere);
status = postIOCompletion(*pPver);
```

Destruction:

When the server library has successfully queued the response message posted by `postIOCompletion()` or when the client disconnects before the asynchronous operation completes, the server library will call the `destroy()` virtual function. destroy() is a member of the `casAsyncIO` class, a class from which `casAsyncPVExistIO` is derived. `destroy()` deletes the `this` pointer, therefore causing the object to be destroyed if the object is on the free store. If the object was not created with `new`, `destroy()` should be redefined in a derived class so that the object is duly destroyed. The server tool should never directly call `destroy()`, should never directly call a `casAsyncPVExistIO` or derived-class destructor, and should never itself delete a `casAsyncPVExistIO` object. This is because a `casAsyncPVExistIO` object should not be destroyed until the server library has successfully queued the response message to the client.

The `casAsyncIO` and casAsync`PVExistIO` classes each have a virtual destructor, `~casAsyncIO()` and `~casAsyncPVExistIO()`. Because the destructors are virtual, they guarantee that the destructors in the casAsync`PVExistIO` class and any derived classes are executed in order. If the server tool derives a class from casAsync`PVExist`IO, it is responsible for providing a destructor that will perform any necessary clean up prior to destruction.

For example, suppose a server tool derived a class called `myAsyncPVExistIO` from `casAsyncPVExistIO` and provided a destructor for the class, `~myAsyncPVExistIO()`. If `a myAsyncPVExistIO` object is created, when the operation completes and the server tool calls `postIOCompletion()`, the server library will queue the response to the client, after which it will call `destroy()`. Assuming that the `myAsyncPVExistIO` object was created using the `new` operator, then the default version of `destroy()` will delete the object causing `~myAsyncPVExistIO()` to be executed before `~casAsyncPVExistIO()` is executed, after which `~casAsyncIO()` is called, after which the object's container is destroyed.

Public Member Functions:

Non-Virtual

```
casAsyncPVExistIO(const casCtx &ctx);
caStatus postIOCompletion(const pvExistReturn &retValIn);
caServer *getCAS();
```

Virtual

```
virtual ~casAsyncPVExistIO();
```

Name:

`casAsyncPVExistIO()`

Synopsis:

```
#include <casdef.h>
casAsyncPVExistIO(const casCtx &ctx);
```

Description:

casAsyncPVExistIO() is the class constructor. The server tool doesn't need to be concerned with what it does, only in passing it the right arguments. Its only argument is a casCtx object, ctx. ctx is used by the server library; the server tool need not be concerned with it, except as far as making sure that it gets passed to the constructor.

Name:

`postIOCompletion()`

Synopsis:

```
#include <casdef.h>
caStatus postIOCompletion(const pvExistReturn &retValIn);
```

Description:

The server tool should call `postIOCompletion()` after it has completed the PV search, passing to it a `pvExistReturn` object which contains `pverExistsHere` or `pverDoesNotExistHere`.

Name:

`getCAS()`

Synopsis:

```
#include <casdef.h>
caServer *getCAS();
```

Description:

`getCAS()` returns a pointer to the `caServer` object associated with the `casAsyncPVExistIO` or derived-class object.

Name:

```
~casAsyncPVExistIO()
```

Synopsis:

```
#include <casdef.h>
virtual ~casAsyncPVExistIO();
```

Description:

~casAsyncPVExistIO() is the class destructor. It is an empty or NULL function, and is only provided to guarantee the execution order of any destructors declared in any derived class or classes. Thus, if a server tool derives a class from the casAsyncPVExistIO class and provides a destructor for that class, when the object is deleted or destroyed, the destructor for the derived class will be called before the destructor of the base class.

The server tool should never call this destructor directly. The server tool should also never directly delete a `casAsyncPVExistIO` object.

CLASS: casAsyncCreatePVIO

Declared: casdef.h

The `casAsyncCreatePVIO` class provides asynchronous completion for `createPV()`. The server library calls `createPV()` when a client requests a connection to a PV and `pvExistTest()` has returned `pverExistsHere`. The task of `createPV()` is to create a PV object (an object derived from the `casPV` class) and to return a pointer to it, to return the object itself (by reference), or to return a status code indicating the error that occurred or indicating that it wants to complete asynchronously. The three possible status codes `createPV()` can return are `S_casApp_pvNotFound`, `S_casApp_noMemory`, and `S_casApp_asyncCompletion`. It should return the last code if it wishes to delay completing its task, in which case it should create a `casAsyncCreatePVIO` object before returning.

The actual return value of `createPV()` is a `pvCreateReturn` object, which is a container for a pointer to a PV object and a status code (of type `caStatus`). Since C++ allows a function to initialize and return an object indirectly to the calling function, when `createPV()` returns one of the above-mentioned values, a `pvCreateReturn` object is properly initialized and returned to the server library. However, asynchronous operations will have to explicitly initialize a `pvCreateReturn` object and pass it to `postIOCompletion()` after the operation completes. The `pvCreateReturn` class has two constructors which can be used to initialize it: one accepts a PV object by reference and the other simply accepts a status code.

```
pvCreateReturn(caStatus statIn)
pvCreateReturn(casPV &pv);
```

With the first constructor, the status code member of the class is initialized to the status code indicated by `statIn`, and the PV pointer is initialized to NULL. With the second constructor, the PV pointer is set to point to the address `&pv`, and the status code is initialized to `S_casApp_success`.

Destruction:

When the server library has successfully queued the response message posted by `postIOCompletion()` or when the client disconnects before the asynchronous operation completes, the server library will call the `destroy()` virtual function. destroy() is a member of the `casAsyncIO` class, a class from which `casAsyncCreatePVIO` is derived. `destroy()` deletes the `this` pointer, therefore causing the object to be destroyed if the object is on the free store. If the object was not created with `new`, `destroy()` should be redefined in a derived class so that the object is duly destroyed. The server tool should never directly call `destroy()`, should never directly call a `casAsyncCreatePVIO` or derived-class destructor, and should never itself delete a `casAsyncCreatePVIO` object. This is because a `casAsyncCreatePVIO` object should not be destroyed until the server library has successfully queued the response message to the client.

The `casAsyncIO` and `casAsyncCreatePVIO` classes each have a virtual destructor, `~casAsyncIO()` and `~casAsyncCreatePVIO()`. Because the destructors are virtual, they guarantee that the destructors in the `casAsyncCreatePVIO` class and any derived classes are executed in order. If the server tool derives a class from `casAsyncCreatePVIO`, it is responsible for providing a destructor that will perform any necessary clean up prior to destruction.

For example, suppose a server tool derived a class called `myAsyncCreatePVIO` from `casAsyncCreatePVIO` and provided a destructor for the class, `~myAsyncCreatePVIO()`. If `a myAsyncCreatePVIO` object is created, when the operation completes and the server tool calls `postIOCompletion()`, the server library will queue the response to the client, after which it will call `destroy()`. Assuming that the `myAsyncPVExistIO` object was created using the `new` operator, then the default version of `destroy()` will delete the object causing `~myAsyncCreatePVIO()` to be executed before `~casAsyncCreatePVIO()` is executed, after which `~casAsyncIO()` is called, after which the object's container is destroyed.

Public Member Functions:

Non-Virtual

```
casAsyncCreatePVIO(const casCtx &ctx);
caStatus postIOCompletion(const pvCreateReturn &retValIn);
caServer *getCAS();
```

Virtual

```
virtual ~casAsyncCreatePVIO();
```

Name:

`casAsyncCreatePVIO()`

Synopsis:

```
#include <casdef.h>
casAsyncCreatePVIO(const casCtx &ctx);
```

Description:

`casAsyncCreatePVIO()` is the class constructor. The server tool doesn't need to be concerned with what it does, only in passing it the right arguments. Its only argument is a `casCtx` object, `ctx`. `ctx` is used by the server library; the server tool need not be concerned with it, except as far as making sure that it gets passed to the constructor.

Name:

`postIOCompletion()`

Synopsis:

```
#include <casdef.h>
caStatus postIOCompletion(const pvExistReturn &retValIn);
```

Description:

The server tool should call `postIOCompletion()` after it has completed the PV search, passing to it a `pvCreateReturn` object which contains a status code and pointer to a PV object. The pointer can be NULL if the status code isn't `S_casApp_success`.

Name:

`getCAS()`

Synopsis:

```
#include <casdef.h>
caServer *getCAS();
```

Description:

`getCAS()` returns a pointer to the `caServer` object associated with the `casAsyncCreatePVIO` or derived-class object.

Name:

`~casAsyncPVExistIO()`

Synopsis:

```
#include <casdef.h>
virtual ~casAsyncPVExistIO();
```

Description:

`~casAsyncCreatePVIO()` is the class destructor. It is an empty or NULL function, and is only provided to guarantee the execution order of any destructors declared in any derived class or classes. Thus, if a server tool derives a class from the `casAsyncCreatePVIO` class and provides a destructor for that class, when the object is deleted or destroyed, the destructor for the derived class will be called before the destructor of the base class.

The server tool should never call this destructor directly. The server tool should also never directly delete a `casAsyncCreatePVIO` object.

CLASS: gdd

The `gdd` class is not actually part of the Portable Server interface. However, both the server library and the functions in the interface use it extensively, so anyone using the interface to write a server must have a basic familiarity with the `gdd` class and its derived classes, `gddAtomic`, `gddScalar`, and `gddContainer`.

Basically, the `gdd` class is a way of encapsulating architecture-independent data so that its characteristics can be described and so that it can be easily exchanged and converted from one type to another. Its core member is a union which has members for each of the architecture-independent types including `aitFixedString` and `aitString`. The union has also has a `void` pointer that can be used to point to an array of whatever type.

A `gdd` object is described by its primitive type and its application type. Its primitive type is its architecture-independent type, i.e., `aitInt8` or `aitFloat32`, etc. Its application type describes what the data is used for and what it represents. For example, a certain application type can describe data as representing an alarm status code or a PV's value or a PV's units. New application types can be defined to describe new uses for data. The `gdd` library comes with pre-defined application types that describe EPICS data.

The following describes some of the member functions of the `gdd` class. Not all of the functions are covered here. The `gdd` class consists of many functions; many of them are meant for internal use. The functions described here are those that are most likely to be of interest to users.

Note that there are three classes that are derived from the `gdd` class: `gddScalar`, which is used for scalar data, `gddAtomic`, which is used for array data, and `gddContainer` which is used to contain multiple `gddAtomic` and/or `gddScalar` objects. For the most part, most of the functions used to deal with `gddScalar` and `gddAtomic` objects are part of the `gdd` base class and not the `gddScalar` or `gddAtomic` objects. For instance, the `gdd` class contains the functions needed to deal with array data. This is meant to support flexibility in dealing with different `gdd` objects. Also note that the following is not an exhaustive treatment of the `gdd` class.

Public Member Functions:

`gdd(),applicationType(),primitiveType(), dimension()`

```
setPrimType(), setApplType(), reset(), clear(),
changeType(), setBound(), getBound(), dataPointer(),
getTimeStamp(),
setTimeStamp(), setStatus(), getStatus(), setStat(),
getSevr(), getStat(), getSevr(), setStatSevr(),
getStatSevr(), copyInfo(), copy(), Dup(), noReferencing(),
reference(), unreference(), isScalar(), isContainer(),
isAtomic(), isManaged(),
isConstant(), isNoRef(), markConstant(), markManaged(),
markUnManaged(), getRef(), putRef(), getConver(),
putConvert(), put(), get(), operator=()
```

Name:

`gdd()`

Synopsis:

```
#include <casdef.h>
gdd(void);
gdd(gdd*);
gdd(int app);
gdd(int app, aitEnum prim);
gdd(int app, aitEnum prim, int dimen);
gdd(int app, aitEnum prim, int dimen, aitUint32*
size_array);
```

Description:

`gdd()` is the class constructor. It is an overloaded consructor with constructors for initializing scalar data and array data, `gddScalar` objects and `gddAtomic` objects. In fact the constructors for the `gddScalar` and `gddAtomic` classes just call these constructors.

Basically, all `gdd` objects have three characteristics: their application type, their primitive type, and their dimensions. For scalar data, the dimensions are zero; for array data, one or above. The first constructor, which accepts no arguments, initializes the the application type to zero, i.e., no application type, the primitive type to aitEnumInvalid, and the dimensions to zero. The second constructor which accepts a pointer to an already initialized `gdd` object initializes the `gdd` object with the characteristics of the already-initialized object.

The third constructor accepts an application type for its sole argument. It initializes the application type to `app`, and then initializes the primitive type to `aitEnumInvalid` and the dimensions to zero.

The fourth constructor initializes the application type to `app`, the primitive type to `prim`, and the dimensions to zero. The fifth constructor does the same except it initializes the dimensions to `dimen`.

For array data, in addition to the dimensions, the size of each dimension can be specified. For example, the sizes of a two dimensional array might be 5 X 10. In order to initialize the size of the array, you must pass an array of `aitUint32` integers as the fifth argument to the last constructor. It must have at least as many elements as array dimensions. The size of each dimension will be initialized to the corresponding element in the array: the first dimension will be initialized with the first element of the array, the second with the second element, and so on.

Note that the last two constructors do not allocate space for an array. The dimensions and the size merely describe the actual array which must be created by the user.

Name:

`applicationType()`

Synopsis:

```
#include <gdd.h>
unsigned applicationType(void) const;
```

Description:

Returns the `gdd` object's application type, an unsigned integer.

Name:

`primitiveType()`

Synopsis:

```
#include <gdd.h>
aitEnum primitiveType(void) const;
```

Description:

Returns the `gdd` object's primitive type, the type of the scalar or array data it contains. `aitEnum` is an enumerated type whose enumerators correspond to the architecture-independent types. In addtion, `aitEnum` has the enumerators `aitEnumInvalid` to describe a `gdd` object which has not been initialized with a primitive type or that contains an unrecognized type, and `aitEnumContainer` to describe a `gddContainer` object.

Name:

```
dimension()
```

Synopsis:

```
#include <gdd.h>
unsigned dimension(void) const;
```

Description:

`dimension()` returns the number of dimensions of an array. If `dimension()` returns zero, the object is a scalar object. If it returns anything greater than zero, the number represents the number of dimensions of the arrayÄone means that the array is a one-dimensional array, two means that the array is a two-dimensional array, and so on.

Name:

```
setPrimType()
```

Synopsis:

```
#include <gdd.h>
void setPrimType(aitEnum t);
```

Description:

A user can use `setPrimType()` to change or initialize the object's primitive type by passing it an enumerator of the `aitEnum` type such as `aitEnumInt8`. Note that changing an object's primitive type does not convert its data.

Name:

```
setApplType()
```

Synopsis:

```
#include <gdd.h>
void setApplType(int t);
```

Description:

`setApplType()` allows the user to change the `gdd` object's existing application type. An application type is an integer constant, a code that describes the object's data and what it representsÄa PV's value, its alarm status, its alarm severity, etc.

Name:

`reset()`

Synopsis:

```
#include <gdd.h>
gddStatus reset(aitEnum primtype, int dimension,
                aitIndex* dim_counts);
```

Description:

`reset()` changes the object's primitive type, the number of its dimensions, and the size of each of the dimension. It sets the primitive type to `primtype`, the number of dimensions to `dimension`, and the size of each dimension it sets to the corresponding element in the array `dim_counts`; it sets the size of the first dimension to element 1, the second to element two, etc.

Note that changing the object's primitive type using `reset()` does not convert the existing data to the new data type. Nor does changing its array dimensions allocate memory for a new array or delete the old array or otherwise perform any operation on the array.

`reset()` will not work for `gddContainer` objects or managed objects.

Name:

`clear()`

Synopsis:

```
#include <gdd.h>
gddStatus clear(void);
```

Description:

For `gddAtomic` or array objects, it destroys the array, and sets the dimensions to zero. It also changes the primitive type to `aitEnumInvalid` and the application type to zero. For `gddContainer` objects, it steps through the container, destroying all the `gdd` objects within. For `gddScalar` objects, it does nothing.

Name:

`changeType()`

Synopsis:

```
#include <gdd.h>
gddStatus changeType(int appltype, aitEnum primtype);
```

Description:

`changeType()` will change the object's application type to `appltype` and the object's primitive type to `primtype`. The function will only work if the object is a scalar object or the primitive type is uninitialized, i.e., `aitEnumInvalid`. Otherwise, the function returns `gddErrorNotAllowed`.

Name:

`setBound()`

Synopsis:

```
#include <gdd.h>
gddStatus setBound(unsigned dim_to_set, aitIndex first,
                   aitIndex count);
```

Description:

`setBound()` lets the user set the bounds of the array's dimensions. The bounds refer to the index of the first element and the element-count of the dimension. For instance, if you wish to describe a two-dimensional array that is 10 X 5, the bounds for the first dimension might be 0 and 10, 0 being the index of the first element, and 10 being the number of elements; and the bounds of the second dimension might be 2 and 5, 2 being the index of the first element and 5 being the number of elements.

The first argument to `setBound()`, `dim_to_set`, indexes the dimension whose bounds are to be set. Note that the dimensions are indexed as they would be in C or C++, i.e., starting at zero. Therefore, to set the bounds of the first dimension, `dim_to_set` should be zero. The second argument, `first`, sets the index of the first element, and the third argument, `count`, sets the element count.

Note that this function is invalid when called for `gddScalar` objects; `gddErrorNotAllowed` is returned when `setBound()` is called for `gddScalar` objects. Also, if `dim_to_set` exceeds *n*-1, where *n* is the number of dimensions in the array, `gddErrorOutOfBounds` is returned.

Name:

`getBound()`

Synopsis:

```
#include <gdd.h>
gddStatus getBound(unsigned dim_to_get, aitIndex& first,
                   aitIndex& count);
```

Description:

`getBound()` allows the user to retrieve the bounds for the dimension indexed by `dim_to_get`. Recall the an array's dimensions are indexed with the first dimension starting at zero. The second and third arguments are passed by reference. The index of the first element is written into first, and the number of elements is written into count.

Note that if `dim_to_get` exceeds *n*-1, where n is the number of dimensions, `gddErrorOutOfBounds` will be returned. In addition, `getBound()` is not supported for `gddScalar` objects.

Name:

`dataPointer()`

Synopsis:

```
#include <gdd.h>
void* dataPointer(void) const;
void* dataPointer(aitIndex element_offset) const;
```

Description:

dataPointer() returns the void pointer of the data member. Recall that a gdd object's data member is a union that has members for all the architecture-independent types as well as a void pointer that can point to arrays for gddAtomic objects. This void pointer can be retrieved by calling dataPointer().

dataPointer() has two forms, one which accepts no arguments and one which accepts an aitIndex argument. The former simply returns the void pointer, which will be the address of the array if the array is initialized and exists. The latter will return the address indexed by the aitIndex argument element_offset. Note that dataPointer() doesn't check to make sure that index does not exceed the array's bounds, so the user must be careful when using the latter form of dataPointer().

Name:

setTimeStamp()

Synopsis:

```
#include <gdd.h>
void setTimeStamp(const struct timespec* const ts);
void setTimeStamp(const aitTimeStamp* const ts);
```

Description:

A gdd object has the ability to store a timestamp, in addition to a piece of architecure-independent data. The user can set a timestamp for the gdd object using setTimeStamp(). setTimeStamp() has two forms; each accepts a pointer to one of two types. The first is a timespec structure for POSIX applications. This structure is defined in gdd.h. The second is an architecture-independent timestamp structure, aitTimeStamp.

Name:

getTimeStamp()

Synopsis:

```
#include <gdd.h>
void getTimeStamp(struct timespec* const ts) const;
void getTimeStamp(aitTimeStamp* const ts) const;
```

Description:

`getTimeStamp()` can be used to retrieve a `gdd` object's timestamp. It has two forms: one which accepts a pointer to `timespec` structure and one which accepts a pointer to `aitTimeStamp` objects. For the `timespec` structure, it will write the appropriate values into the structure. Thus, memory must already be allocated for `ts`. For the pointer to an `aitTimeStamp` object, the pointer is merely set to point to the object's aitTimeStamp object. Thus, `ts` does not have to point to an existing aitTimeStamp object, i.e., it can be NULL.

Note that `getTimeStamp()` will only work if `setTimeStamp()` has already been called. Also note that the type of argument passed to `getTimeStamp()` must match the type passed to `setTimeStamp()`. For instance, if `setTimeStamp(const struct timespec* const ts)` is called, then `getTimeStamp(struct timespec* const ts)` must be called to retrieve the timestamp and not `getTimeStamp(aitTimeStamp* const ts)`.

Name:

`setStatus()`

Synopsis:

```
#include <gdd.h>
void setStatus(aitUint32);
void setStatus(aitUint16 high, aitUint16 low);
```

Description:

A `gdd` object has the ability to store a status value, in addition to architecture-independent data. Status values stem from EPICS alarm status values, but they don't necessarily have to be EPICS-related, as long as the values are positive integers. `setStatus()` can be used to write these values into the `gdd` object.

`setStatus()` has two forms. The first sets a single value of type `aitUint32`; the second actually sets two values of type `aitUint16`, a high value and a low value. In addition, a `gdd` object can also hold a value which represents the alarm severity. However, in order for a `gdd` object to hold both, it must call `setStat()` and `setSevr()`, or `setStatSevr()`. Calling `setStatus()` and then `setSevr()` will cause part of the status value to be overwritten.

Name:

```
getStatus()
```

Synopsis:

```
#include <gdd.h>
void getStatus(aitUint32&);
void getStatus(aitUint16& high, aitUint16& low);
```

Description:

`getStatus()` retrieves an object's status. All `gdd` objects have the ability to store a status value, which is an unsigned integer that represents, in EPICS applications at least, the status of an alarm condition. There are two forms: one which retrieves a single value of type `aitUint32` and one which retrieves two values of type `aitUint32`, a high status value and a low status value. To call the first form, pass an lvalue of type `aitUint32` to `getStatus()`. To call the second form, pass two lvalues of type `aitUint16`, where the first lvalue will hold the high value, and the second, the low value. Note the arguments of both forms are passed by reference, so the proper values will be written into the arguments.

Note that `getStatus()` is only valid if `setStatus()` has been called previously at least once. If `setStatus()` hasn't been called, the values written into the arguments will be garbage. Also not that the form of `getStatus()` called must correspond to the form of `setStatus()` what was called. For instance, if `setStatus(aitUint32)` is called, the user must call `getStatus(aitUint32&)`, not `getStatus(aitUint16& high, aitUint16& low)`. If the wrong form is called, the result will not be unpredictable or erroneous results.

Name:

```
setStat()
```

Synopsis:

```
#include <gdd.h>
void setStat(aitUint16);
```

Description:

A `gdd` object can hold both an alarm status value and an alarm severity value. However, if the user intends to hold both the status and the severity. It must call `setStat()` and then `setSevr()`, or else `setStatSevr()`. Calling `setStatus()` and then `setSevr()` or the other way around will cause the first value to be overwritten.

`setStat()` accepts a value of type `aitUint16`. It will set the alarm status to this value. If you set the status using `setStat()` instead of `setStatus()`, you must retrieve the value using either `getStat()` or `getStatSevr()`.

Name:

`getStat()`

Synopsis:

```
#include<gdd.h>
aitUint16 getStat(void) const;
```

Description:

`getStat()` can be used to retrieve an object's status value that was written using `setStat()` or `setStatSevr()`. It should not be used to retrieve the status value when the value was written using `setStatus()`.

Name:

`setSevr()`

Synopsis:

```
#include <gdd.h>
void setSevr(aitUint16 s);
```

Description:

With `setSevr()`, the user can set the severity of a `gdd` object. Simply pass a value of type `aitUint16` as an argument.

Name:

`getSevr()`

Synopsis:

```
#include <gdd.h>
aitUint16 getSevr(void) const;
```

Description:

`getSevr()` returns the object's alarm severity, written into the object using either `setSevr()` or `setStatSevr()`.

Name:

`setStatSevr()`

Synopsis:

```
#include <gdd.h>
void setStatSevr(aitInt16 stat, aitInt16 sevr);
```

Description:

With `setStatSevr()` the user can set the severity and status for the object by passing to it two arguments of type `aitInt16`, where the first is the status and the second the severity. The status and severity can retrieved using `getStat()`, `getSevr()`, or `getStatSevr()`.

Name:

`getStatSevr()`

Synopsis:

```
#include <gdd.h>
void getStatSevr(aitInt16& stat, aitInt16& sevr)
```

Description:

`getStatSevr()` can be used to retrieve the status and severity by passing to it two lvalues of type `aitInt16`. It will then write the appropriate values into each lvalue, as long as the values were written using `setStat()`, `setSevr()`, or `setStatSevr()`.

Names:

`copyInfo(), copy(), Dup()`

Synopsis:

```
#include <gdd.h>
gddStatus copyInfo(gdd*);
gddStatus copy(gdd*);
gddStatus Dup(gdd*);
```

Description:

These three functions can be used to copy the characteristics and data from another `gdd` object to the object for which they are called. For instance, if `thatObject` is a `gddScalar` object, whose primitive type is `aitEnumUint16`, whose application type is "severity", and whose value is `197`, then if `copyInfo`() is called for the `gdd` object `thisObject`, `thatObject`'s application type, primitive type, and value will be copied to `thisObject` so that `thisObject` will have the primitive type `aitEnumUint16`, the application type "severity," and the value `197`.

All these functions accept the same argument, a pointer to a `gdd` object which can be a `gddContainer` object, a `gddScalar` object, or a `gddAtomic` object.

When the pointer points to a `gddContainer` object, all three functions have the same results. They copy the primitive type, the application type, and all the references to the `gdd` objects contained within. Thus, the contained objects are not actually reproduced, but instead the references to them are copied from the referenced object.

When the pointer points to a `gddScalar`, all three functions also have the same results. They copy the primitive type, the application type, and the data contained in the data member from the referenced object to the object for which the function is called.

When the pointer points to a `gddAtomic` object on the other hand, the three functions have different results. `CopyInfo()` will copy the information about the object, its characteristics, but will not copy the array from the referenced object to the object for which `copyInfo()` is called; that is, it will copy the primitive type, the application type, the dimensions, and the bounds of the dimensions, but not the actual array. `copy()` does the same things as `copyInfo()` except that it allocates space for an array and copies the array data from the referenced object to the object for which `copy()` was called. `Dup()`, on the other hand, does the same thing as `copy()`, but instead of copying the array data, it merely references it with a pointer.

Name:

```
isScalar(), isAtomic(), isContainer()
```

Synopsis:

```
#include <gdd.h>
int isScalar(void) const;
int isAtomic(void) const;
int isContainer(void) const;
```

Description:

These functions can be called to find out whether the object is a `gddScalar`, `gddAtomic`, or `gddContainer` object. For instance, `isScalar()` returns a non-zero value if the object is a `gddScalar` object or a zero if the object is not a `gddScalar` object. Each returns a non-zero, zero if the object is, is not the specified object type.

The conditions which distinguish one object from another are simple: if the dimensions are zero the object is a `gddScalar` object; if the primitive type is `aitEnumContainer`, the object is a `gddContainer` object; if the dimensions are greater than 0 and the primitive type is not `aitEnumContainer`, the object is a `gddAtomic` object.

Names:

```
NoReference(), Reference(), Unreference()
```

Synopsis:

```
#include <gdd.h>
gddStatus NoReferencing(void);
gddStatus Reference(void);
gddStatus Unreference(void);
```

Description:

Each `gdd` object has a counter that indicates how many times it is being referenced. When an object is created, this counter is incremented by one. For each additional pointer that is set to point to the object, `Reference()` should be called. Each time that a pointer to a `gdd` object is set to point somewhere else or is no longer needed, `Unreference()` should be called, this includes the original pointer to the object. This goes for objects not on the free store. `Unreference()` should be called when the object is no longer needed.

`Reference()` causes the counter to be incremented by one. Since the counter is automatically incremented when an object is first created, it doesn't have to be incremented for the original reference. `Unreference()` causes the counter to be decremented. When the counter reaches zero, `gddDestructor::Destroy()` is called. `Destroy()` calls `gddDestructor::Run()`. `Run()` is a virtual function that is meant to destroy or clean up the object's data. It is passed a pointer to the object's data. By default `Run()` does a `delete []` on an array of `aitInt8` elements. The user should redefine `Run()` if the data contained in the `gdd` object is of a different type. Note that no cleanup is necessary for scalar data.

By calling `Noreferencing()` for an object, the user can disallow any any references to the gdd object other than the current one provided that the reference count is no greater than one, i.e., the object hasn't already been referenced.

Name:

`markConstant(), markManaged(), markUnmanaged()`

Synopsis:

```
#include<gdd.h>
void markConstant(void);
void markManaged(void);
void markUnmanaged(void);
```

Description:

These three somewhat unrelated functions all mark the `gdd` object with a certain characteristic. None accepts any arguments or returns any value. `markConstant`() marks the data as constant, meaning that it cannot be changed in its lifetime.

`markManaged()` marks the object as a managed object. Usually, a managed object is a `gddContainer` object that has a predefined structure, i.e., a predefined number of contained objects in a predefined order. Examples of managed containers are those mapped to DBR structures. Clients can make requests using data structures. The general idea is that the server will write a set of attributes into the structure as well as the PV's value. For instance, the DBR_STS structure can be used to request a PV's value as well as its alarm status and severity. These structures can be mapped to a `gddContainer` object. They are marked as managed to indicate that they have a predefined mapping or structure.

markUnmanaged() will reverse markManaged().

Name:

`getRef()`

Synopsis:

```
#inlcude <gdd.h>
void getRef(aitFloat64*& d);
void getRef(aitFloat32*& d);
void getRef(aitUint32*& d);
void getRef(aitInt32*& d);
void getRef(aitUint16*& d);
void getRef(aitInt16*& d);
void getRef(aitUint8*& d);
void getRef(aitInt8*& d);
void getRef(aitString*& d);
void getRef(aitFixedString*& d);
void getRef(void*& d);
```

Description:

`getRef()` is used to reference a `gdd` object's data member. Remember that an object's data member is a union which is typedefed as `aitType`. The declaration of this union can be found in aitTypes.h. `getRef()` typecasts the union according to the pointer passed to it, thus accessing the correct member. Note that the pointer is passed by reference so that the correct address is directly assigned to it. Also note that the typecasting a gdd object will have the same effect, as in

```
aitInt8* pValue = (aitInt8*)gDDObject;
```

where `gDDObject` is the gdd object whose data is being referenced.

Since the the value is being referenced using a pointer, it is possible that `gDDObject` may be destroyed while its data is being referenced. To keep `pValue` from pointing to a non-existant piece of data, `Reference()` should be called after `getRef()` is called. However, `Unreference()` will have to be called for the object whose data is referenced, after the pointer is no longer needed.

Name:

`putRef()`

Synopsis:

```
#include <gdd.h>
void putRef(void* v,aitEnum code, gddDestructor* d = NULL);
void putRef(aitFloat64* v, gddDestructor* d = NULL);
void putRef(aitFloat32* v, gddDestructor* d = NULL);
void putRef(aitUint8* v, gddDestructor* d  = NULL);
void putRef(aitInt8* v, gddDestructor* d = NULL);
void putRef(aitUint16* v, gddDestructor* d = NULL);
void putRef(aitInt16* v, gddDestructor* d = NULL);
void putRef(aitUint32* v, gddDestructor* d = NULL);
void putRef(aitInt32* v, gddDestructor* d = NULL);
void putRef(aitString* v, gddDestructor* d = NULL);
void putRef(aitFixedString* v, gddDestructor* d = NULL);
void putRef(const aitFloat64* v, gddDestructor* d = NULL);
void putRef(const aitFloat32* v, gddDestructor* d = NULL);
void putRef(const aitUint8* v, gddDestructor* d  = NULL);
void putRef(const aitInt8* v, gddDestructor* d = NULL);
void putRef(const aitUint16* v, gddDestructor* d = NULL);
void putRef(const aitInt16* v, gddDestructor* d = NULL);
void putRef(const aitUint32* v, gddDestructor* d = NULL);
void putRef(const aitInt32* v, gddDestructor* d = NULL);
void putRef(const aitString* v,gddDestructor* d=NULL);
void putRef(const aitFixedString* v,gddDestructor* d=NULL);
```

Description:

`putRef()` allows a user to set the void pointer of a `gdd` object's data member to point to an "outside" piece of data, which can be either an array or a scalar value. `putRef()` accepts a pointer to any of the architecture-independent types. If the pointer is a constant pointer, the object will be marked as constant, i.e., its data cannot be changed through the gdd class.

`putRef()` allows a pointer to a `gddDestructor` to be passed as the second argument, though this argument is optional. The idea here is that the object that will be referenced data will be destroyed when the object referencing it is destroyed.

Name:

`getConvert()`

Synopsis:

```
#include <gdd.h>
void getConvert(aitFloat64& d);
void getConvert(aitFloat32& d);
void getConvert(aitUint32& d);
void getConvert(aitInt32& d);
void getConvert(aitUint16& d);
void getConvert(aitInt16& d);
void getConvert(aitUint8& d);
void getConvert(aitInt8& d);
void getConvert(aitString& d);
void getConvert(aitFixedString& d);
```

Description:

`getConvert()` allows the user to retrieve an object's data in a form that may be different from the object's primitive type. `getConvert()` accepts any lvalue of an architecture-independent type. It will perform the proper conversion of the data and write it into the lvalue.

Name:

`putConvert()`

Synopsis:

```
#include <gdd.h>
void putConvert(aitFloat64 d);
void putConvert(aitFloat32 d);
void putConvert(aitUint32 d);
void putConvert(aitInt32 d);
void putConvert(aitUint16 d);
void putConvert(aitInt16 d);
void putConvert(aitUint8 d);
void putConvert(aitInt8 d);
void putConvert(aitString d);
void putConvert(aitFixedString& d);
```

Description:

`putConvert()` accepts a value and writes it into the `gdd` object's data member. If the value does not match the object's primitive type, it will convert the value to the object's primitive type. It cannot be called for `gddContainer` or `gddAtomic` objects.

Name:

`put()`

Synopsis:

```
#include <gdd.h>
gddStatus put(const aitFloat64* const d);
gddStatus put(const aitFloat32* const d);
gddStatus put(const aitUint32* const d);
gddStatus put(const aitInt32* const d);
gddStatus put(const aitUint16* const d);
gddStatus put(const aitInt16* const d);
gddStatus put(const aitUint8* const d);
gddStatus put(const aitInt8* const d);
gddStatus put(const aitString* const d);
gddStatus put(const aitFixedString* const d);
gddStatus put(const gdd* dd);
```

Description:

This form of `put()` can accept a pointer to any architecture-independent type or a pointer to a `gdd` object. The results depend on the type of object for which `put()` is called and the data type to which the pointer `d` points.

If the object for which `put()` was called is a scalar object and `d` points to an architecture-independent scalar value, then `*d` will be written into the the object's data member. If `d` points to an array, only the first element will be written.

If the object for which `put()` was called is an atomic object (`gddAtomic`) and `d` points to an architecture-independent array or scalar value, then *n* elements will be copied into the array of the object for which `put()` was called, where *n* is the smallest element count of either array, or 1 if `d` points to a scalar value. For example, if `d` points to a five-element array and `put()` is called for a `gddAtomic` object which contains a ten-element array, then five-elements will be copied from d to the `gddAtomic` object for which `put()` was called. If d points to a scalar value, then it will be copied into the first element of the `gddAtomic` object. If `put()` is called for a `gddAtomic` object which already has memory allocated for an array that may or may not contain values, *n* values of the array will be overwritten starting with the first element, where *n* is the number of elements written into the array. If `put()` is called for a `gddAtomic` object which doesn't already have memory allocated for its array, memory will be allocated and will be written into. However, the `gddAtomic` object must have its dimensions and bounds initialized in order for memory to be allocated. Instead, `put()` may treat such an object as a scalar value.

The last function, `put(const gdd* dd)`, works a little bit differently from the other functions. It cannot be called for a `gddContainer` object, and its argument, `dd`, cannot point to a `gddContainer` object. If the destination object and the source object are both `gddAtomic` objects, then the source object's array will be copied to the destination object's array. However, `put()` will not copy the array if the source object's array is not a one-dimensional array, for instance, if it's a two- or three-dimensional array, in which case `put()` will return `gddErrorNotSupported`. In addition, the `put()` will fail if the source array has a greater number of elements from the destination object, in which case `put()` will return `gddErrorOutOfBounds`.

The source object can be a scalar object or the destination object can be a scalar object. If both are scalar objects, `put(const gdd* dd)` the data member from the source object is simply written into the data member of the destination object. If the source object is a `gddAtomic` object and the destination object a `gddScalar` object, the first element from the source array will be copied into the data member of the destination object. If the source object is a `gddScalar` object and the destination object a `gddAtomic` object, the source object's scalar value will be written into the first element of the destination object's array.

Name:

`put()`

Synopsis:

```
#include <gdd.h>
void put(aitFloat64 d);
void put(aitFloat32 d);
void put(aitUint32 d);
void put(aitInt32 d);
void put(aitUint16 d);
void put(aitInt16 d);
void put(aitUint8 d);
void put(aitInt8 d);
void put(aitString d);
void put(aitFixedString& d);
void put(aitType* d);
```

Description:

This form of `put()` doesn't return a `gddStatus` value. Except for the last function, `put(aitType* d)`, each `put()` accepts an architecture-independent scalar value and writes it into the `gdd` object for which `put()` was called.

The last function, `put(aitType* d)`, accepts a pointer to an `aitType` union. It will then assign the value from the union `d` to the object's own union.

Name:

```
operator=()
```

Synopsis:

```
#include <gdd.h>
gdd& operator=(const gdd& v);
gdd& operator=(aitFloat64* v);
gdd& operator=(aitFloat32* v);
gdd& operator=(aitUint32* v);
gdd& operator=(aitInt32* v);
gdd& operator=(aitUint16* v);
gdd& operator=(aitInt16* v);
gdd& operator=(aitUint8* v);
gdd& operator=(aitInt8* v);
gdd& operator=(aitString* v);
gdd& operator=(aitFixedString* v);
gdd& operator=(aitFloat64 d);
gdd& operator=(aitFloat32 d);
gdd& operator=(aitUint32 d);
gdd& operator=(aitInt32 d);
gdd& operator=(aitUint16 d);
gdd& operator=(aitInt16 d);
gdd& operator=(aitUint8 d);
gdd& operator=(aitInt8 d);
gdd& operator=(aitString d);
```

Description:

The overloaded `operator=()` functions provide a simple way to copy information from one `gdd` object to another and to write data into arrays. When a `gdd` object appears on the right side of an assignment expression, `operator=(const gdd& v)` is invoked, the contents of the `gdd` source object are copied to the `gdd` destination object. However, only the container is actually copied. The data referenced by the source object is not copied, though the references are.

When a pointer to an architecture-independent value appears on the right side of the operator, the `operator=()` functions like `operator=(aitFloat64* v)` are invoked. These functions achieve the same result as calling `putRef()` for the same architecture-independent type; that is, the data that appears on the right side of the assignment `operator=()`, is referenced by the `void` pointer in the data member of the `gdd` object on the left side of the operator.

When a value of an architecture-independent type appears on right side of the assigment operator, the `operator=()` functions like `operator=(aitFloat64 d)` are invoked. These accomplish the same results as the `void put()` functions (different from the gddStatus `put()` functions). In this case, the value is written into the data member of the source object, the object that appears on the left side of the assignment operator.

Name

`get()`

Synopsis:

```
#include <gdd.h>
void get(void* d);
void get(aitFloat64* d);
void get(aitFloat32* d);
void get(aitUint32* d);
void get(aitInt32* d);
void get(aitUint16* d);
void get(aitInt16* d);
void get(aitUint8* d);
void get(aitInt8* d);
void get(aitString* d);
void get(aitFixedString* d);
void get(aitFloat64& d);
void get(aitFloat32& d);
void get(aitUint32& d);
void get(aitInt32& d);
void get(aitUint16& d);
void get(aitInt16& d);
void get(aitUint8& d);
void get(aitInt8& d);
void get(aitString& d);
void get(aitFixedString& d);
void get(aitType& d);
```

Description:

`get()` is basically used to retrieve a value from the gdd object for which it is called. It has two basic forms. One form accepts a pointer to an architecture-independent array or scalar value and the other form accepts either an architecture-independent value or an `aitType` union.

The functions which accept a pointer are made to retrieve an array of values from a `gddAtomic` object. The values from the `gddAtomic` object are written into the array pointed to by `d`. Note, however, that `d` should point to an array which contains at least as many elements as the `gddAtomic` object's array. If `d` points to an array of an architecture-independent type as in `get(aitInt8* d)`, then the values from the `gddAtomic` object's array will be converted to the architecture-independent type of the destination array. For example, if `get(aitInt8* d)` is invoked, the values of the array elements from the `gddAtomic` object will be converted to aitInt8 values. When the type d points to is undefined, `get(void* d)` is called, which copies the elements from the `gddAtomic` array to `d` without converting the values, in which case `d` will point to an array of values which are the same type as the primitive type of the `gddAtomic` object.

The `get()` functions which accept an architecture-independent lvalue are used to retrieve values from `gddScalar` objects. The value from the data member of the source object, the object for which the `get()` function was called, will be written into the lvalue passed to the function. For instance, if `get(aitFloat64& d)` is called, the `Float64` member of the data union will be written into `d`.

CLASS: gddAtomic

Declared: gdd.h

The `gddAtomic` class is a class derived from the `gdd` class. It is used to store arrays. Except for the `gddAtomic` constructors and the `operator=()` functions, there are no other functions currently defined for it. All functions needed to deal with arrays are part of the `gdd` class. Event the constructor or constructors simply call `gdd` class constructors in order to initialize a `gddAtomic` object.

`isAtomic()` will return True if any `gdd` object has dimensions greater than zero and the object is not a `gddContainer` object. Thus, it's possible to create a `gdd` object not using the `gddAtomic` class that can contain arrays if the proper constructors are called. However, this is not recommended.

See the `gdd` class for more information on the functions that can be used with gddAtomic objects.

Name:

```
gddAtomic()
```

Synopsis:

```
        #include <gdd.h>
        gddAtomic(void);
        gddAtomic(gddAtomic* ad);
gddAtomic(int app);
gddAtomic(int app, aitEnum prim);
gddAtomic(int app, aitEnum prim, int dimen,
        aitUint32* size_array);
gddAtomic(int app, aitEnum prim, int dimen, ...);
```

Description:

The `gddAtomic` constructors can be used to initialize a `gddAtomic` object. However, none of them can be used to allocate memory for an array or otherwise create an actual array. The most any do is to define the dimensions and the bounds of the array.

The first constructor, `gddAtomic(void)`, accepts no arguments an merely creates the object. The application type, primitive type, dimensions, and bounds of the array will have to be initialized subsequently. The second constructor, `gddAtomic(gddAtomic* gdd)`, accepts a pointer to `gddAtomic` object. It will initialize the object with the characteristics of the object pointed to by the pointer.

The third constructor accepts an application type for its sole argument. It initializes the application type to `app`, and then initializes the primitive type to `aitEnumInvalid` and the dimensions to zero.

The fourth constructor initializes the application type to `app`, the primitive type to `prim`, and the dimensions to zero. The fifth constructor does the same except it initializes the dimensions to `dimen`.

For array data, in addition to the dimensions, the size of each dimension can be specified. For example, the sizes of a two dimensional array might be 5 X 10. In order to initialize the size of the array, you must pass an array of `aitUint32` integers as the fifth argument to the this constructor. It must have at least as many elements as array dimensions. The size of each dimension will be initialized to the corresponding element in the array: the first dimension will be initialized with the first element of the array, the second with the second element, and so on.

As an alternative to the last constructor, instead of passing an array of integers to specify the bounds, for each dimension you can simply pass an integer to specify the bounds. Thus, in order to specify the bounds for a three dimensional array, you can simply pass three integers after dimen. The first integer determines the bounds of the first dimension, the second, of the second dimension, and the third, of the third dimension.

Name:

`operator=()`

Synopsis:

```
#include <gdd.h>
gddAtomic& operator=(aitFloat64* v)
gddAtomic& operator=(aitFloat32* v)
gddAtomic& operator=(aitUint32* v)
gddAtomic& operator=(aitInt32* v)
gddAtomic& operator=(aitUint16* v)
```

```
gddAtomic& operator=(aitInt16* v)
gddAtomic& operator=(aitUint8* v)
gddAtomic& operator=(aitInt8* v)
```

Description:

The `operator=()` functions accept a pointer to an architecture-independent type. They will set the void pointer of the `gddAtomic` object's to point to the location pointed to by `v`. `v` is assumed to point to an array, but it can point to a scalar value.

CLASS: gddScalar

Declared: gdd.h

The `gddScalar` class is actually derived from the `gddAtomic` class. In fact, it is just a `gddAtomic` class whose dimension is zero. And as with the `gddAtomic` class, most of the functions needed to deal with scalar values are part of the `gdd` base class. The only functions specifically defined for the `gddScalar` class are the constructors and the `operator=()` functions.

Name:

`gddScalar()`

Synopsis:

```
#include <gdd.h>
gddScalar(void);
gddScalar(gddScalar* ad);
gddScalar(int app);
gddScalar(int app,aitEnum prim);
```

Description:

The `gddScalar()` constructors can be used to create `gddScalar` objects. Once again, a `gddScalar` object is a `gdd` object whose dimensions are zero; that is, if `gdd::dimension()` is called for a `gddScalar` object, a zero should be returned.

The first constructor, `gddScalar(void)`, accepts no arguments and initializes the object's application type to zero (no application type), the object's primitive type to `aitEnumInvalid`, and the object's dimensions to zero.

The second constructor, `gddScalar(gddScalar* ad)`, accepts a pointer to another `gddScalar` object, `ad`, as its only argument and intializes the object with the characteristics of ad.

The third constructor, `gddScalar(int app)`, accepts an application type as its only argument and initializes the object's application type to zero. It initializes the object's primitive type to `aitEnumInvalid` and the object's dimensions to zero.

The fourth constructor, `gddScalar(int app, aitEnum prim)`, initializes the `gddScalar` object's application type to `app`, its primitive type to `prim`, and its dimensions to zero.

Name:

`operator=()`

Synopsis:

```
#include <gdd.h>
gddScalar& operator=(aitFloat64 d);
gddScalar& operator=(aitFloat32 d);
gddScalar& operator=(aitUint32 d);
gddScalar& operator=(aitInt32 d);
gddScalar& operator=(aitUint16 d);
gddScalar& operator=(aitInt16 d);
gddScalar& operator=(aitUint8 d);
gddScalar& operator=(aitInt8 d);
```

Description:

The `gddScalar::operator=()` function is invoked whenever a `gddScalar` object appears on the righthand side of the assigment operator and an architecture independent value appears on the left hand side. The value is then written into the object's data member.

CLASS: gddContainer

Declared: gdd.h

The `gddContainer` class is derived directly from the `gdd` base class. It is used to contain other `gdd` objects. An example of the use of the `gdd` class is for mapping compound types to `gdd` objects. A Channel Access client can make a request using one of several database request types. Some of these types are simple types such as `DBR_FLOAT` that can simply retrieve a PV's value. Others are compound types such as `DBR_CTRL_FLOAT` that consist of a data structure which has members for the PV's value as well as several of its attributes such as its units, alarm status, control limits, etc.

All `DBR` types are mapped to a `gdd` object when the Portable Server receives the request. Simple reqeust types such as `DBR_FLOAT` are mapped into a single `gdd` object since they are a reqeust for a single value. However, compound request types consists of several values and can't be mapped into a single `gdd` object. Therefore, they are mapped into many objects which are contained in a `gddContainer` object. The `gddContainer` object will have an application type that describes what `DBR` request type it's for. For example, the application type for a `gddContainer` that has been mapped for the request type `DBR_CTRL_FLOAT` is represented by the constant `gddAppType_dbr_ctrl_float`.

Currently, the client-side API only allows read or get operations to request compound types. That is, a client can reqeust to read a PV's value using `DBR_CTRL_FLOAT` or some other compound type, but it cannot write to the PV's value using `DBR_CTRL_FLOAT` or any other compound type. Write opertions can only use simple `DBR` types. Therefore, currently, a server tool only need prepare to read a `gddContainer` object, not write one.

The `gddContainer` class has members to add and remove objects. It also has a way to step through it, accessing all its members. However, the `gddAppFuncTable<PV>` class is provided to help server tools perform read operations on a `gddContainer` object. Thus, it's not necessary for a server tool to directly access a `gddContainer` object. See the `gddAppFuncTable` class for more information.

CLASS: aitString

Declared: aitHelpers.h

The `aitString` class is an architecture-independent way to deal with strings. It is a simple class which contains a pointer to a string, members for initializing the string, deleting it, and for keeping track of and accessing attributes such as its length. The `aitString` class can contain two types of strings, constant strings and "mallocated" strings. For the former type, the `aitString` merely references the string. For the latter type, the `aitString` object creates and references its own array of `chars`. If an `aitString` object is initialized with a string when it is created, its string is constant. Afterwards if the string is installed in the object, the string is automatically not constant. Not all functions of the `aitString` class are presented here.

Destruction:

The `aitString` class has no destructors. To provide clean-up before the object is destroyed, the application can make an explicit call to `clear()`. `clear()`, however, only does a `delete []` on mallocated strings, not on constant strings. Remember that the `aitString` only references constant strings, so it provides no method to delete such strings.

Public Member Functions:

```
aitString(void)
aitString(const char* x)
aitString(char* x)
aitUint32 length(void)
void clear(void)
const char* string(void)
aitString& operator=(const char* p)
aitString& operator=(char* p)
int installString(const char* p)
int installString(char* p)
```

```
static aitUint32 totalLength(aitString* array,
                      aitIndex arraySize)
static aitUint32 stringsLength(aitString* array,
                      aitIndex arraySize)
```

Name:

```
aitString()
```

Synopsis:

```
#include <aitTypes.h>
aitString(void);
aitString(const char* x);
aitString(char* x);
```

Description:

There are three `aitString` constructors. The first form of the function, `aitString(void)`, allows an application to create an `aitString` object without specifying a string to initialize it with, thus making the contained string a NULL string. Then, a string can be written into the `aitString` object, in which case the string will be a non-constant string. For all `aitString` objects that are going to deal with non-constant strings, they should use `aitString(void)`, that is, they should not pass a pointer to the constructor when the object is created.

The other two constructors, `aitString(const char* x)` and `aitString(char* x)`, will initialize the `aitString` object with the string, in which case the string will be a constant string. For constant strings, the `aitString` object merely references the string, i.e., it doesn't have its own copy of the string. Remember that the `clear()` cannot be called for constant strings.

Name:

```
length()
```

Synopsis:

```
#include <aitTypes.h>
aitUint32 length(void);
```

Description:

`length()` returns the length of the string. It returns the length for constant as well as non-constant strings.

Name:

`clear()`

Synopsis:

```
#include <aitTypes.h>
void clear(void);
```

Description:

`clear()` will delete non-constant strings. `clear()` must be called explicitly, though it is called by `installString()` to delete the old string before the new string is written.

Name:

`string()`

Synopsis:

```
#include <aitTypes.h>
const char* string(void) const;
```

Description:

`string()` returns the string contained in the `aitString` object, that is, it returns a pointer-to-char, unless the string is NULL, in which case `string()` returns NULL.

Name:

`operator=()`

Synopsis:

```
#include <aitTypes.h>
aitString& operator=(const char* p);
aitString& operator=(char* p);
```

Description:

The `operator=()` functions are invoked when an `aitString` object appears on the left-hand side of the assignment operator and when a character string, either constant or non-constant, appears on the right. The old string contained in the `aitString` object is cleared, and the `new` string pointer is referenced as a constant string, regardless of whether or not the assigned string is constant or non-constant. If the old string is a constant string, it is dereferenced and the string pointer points to the new string.

Name:

`installString()`

Synopsis:

```
#include <aitTypes.h>
int installString(const char* p);
int installString(char* p);
```

Description:

`installString()` can be used to write a new string into an `aitString` object. If the object already contains a string, it will be deleted if a non-constant string, and the new string will be copied into the `aitString` object, so it will be a mallocated, non-constant string.

Name:

`totalLenth()`

Synopsis:

```
#include <aitTypes.h>
static aitUint32 totalLength(aitString* array,
                                                 aitIndex
arraySize);
```

Description:

This is a static, public function. It returns the total bytes used by an array of aitStrings. It accepts an array of `aitString` objects and an `aitIndex` value indicating the size of that array.

Name:

```
stringsLength()
```

Synopsis:

```
#include <aitTypes.h>
static aitUint32 totalLength(aitString* array,
                                                       aitIndex
arraySize);
```

Description:

`stringsLength()` returns the total combined lengths of all the strings in an array of strings, including the NULL character of each string. It accepts an array of `aitString` objects and an `aitIndex` value indicating the length of the array.

CLASS: aitTimeStamp

Declared: aitHelpers.h

The aitTimeStamp class is an architecture-independent way to represent a time stamp. It is used by the `gdd` library and can be used with the `osiTime` class. Basically, it has two `unsigned long` members, one to represent the seconds and one the nanoseconds. Each of these members can be retrieved individually, or a floating-point value can be retrieved where

```
floating_point_value = nanoseconds + seconds
```

Thus, the floating-point value will equal the number of seconds plus the fractional part represented by the nanoseconds.

Constructors are provided to give the `aitTimeStamp` object initial values for its second and nanosecond members, after which the + and - operators can be used to change the values of the second and nanosecond members. In addition, the >= operator has been overloaded to allow an application to compare two `aitTimeStamp` objects.

Destruction:

The `aitTimeStamp` needs no destructor. If an application derives a class from the aitTimeStamp class, it alone is responsible for providing a destructor if necessary. The aitTimeStamp object will be destroyed as all other classes in C++.

Public Member Functions:

```
aitTimeStamp ()
aitTimeStamp (const aitTimeStamp &t)
aitTimeStamp (const unsigned long tv_secIn,
              const unsigned long tv_nsecIn)
friend aitTimeStamp operator+ (const aitTimeStamp &lhs,
                      const aitTimeStamp &rhs)
friend aitTimeStamp operator- (const aitTimeStamp &lhs,
                      const aitTimeStamp &rhs)
friend int operator>=(const aitTimeStamp &lhs,
                   const aitTimeStamp &rhs)
void get(unsigned long &tv_secOut, unsigned long
&tv_nsecOut)
operator double()
operator float()
```

Name:

`aitTimeStamp()`

Synopsis:

```
#include <aitTypes.h>
aitTimeStamp ();
aitTimeStamp (const aitTimeStamp &t);
aitTimeStamp(const unsigned long tv_secIn,
             const unsigned long tv_nsecIn);
```

Description:

There are three `aitTimeStamp` constructors. The first, `aitTimeStamp()`, accepts no arguments. It initializes the seconds and nanoseconds of the time stamp to zero. The second, `aitTimeStamp(const aitTimeStamp &t)`, accepts another `aitTimeStamp` object, `t`, and will initialize the object with the seconds and nanoseconds of `t`. The third, `aitTimeStamp(const unsigned long tv_secIn, const unsigned long tv_nsecIn)`, allows the application to give the `aitTimeStamp` object initial values for its seconds and nanoseconds members.

Name:

`operator+()`

Synopsis:

```
#include <aitTypes.h>
friend aitTimeStamp operator+ (const aitTimeStamp &lhs,
                      const aitTimeStamp &rhs);
```

Description:

The `operator+()` function is invoked when an `aitTimeStamp` object appears on the right and left sides of the addition (+) operator. The seconds from the object on the left are added to the seconds from the object on the right; likewise, the nanoseconds from the lefthand object are added to the nanoseconds of the righthand object. The `aitTimeStamp` object that is assigned the value of the operation is assigned these two sums.

Name:

`operator-()`

Synopsis:

#include <aitTypes.h>

```
friend aitTimeStamp operator-(const aitTimeStamp &lhs,
                              const aitTimeStamp &rhs);
```

Description:

The `operator-()` function is invoked when an `aitTimeStamp` object appears on the righthand and lefthand sides of the subtraction operator (-) in an assignment expression. If the seconds of the righthand object are less than the seconds of the lefthand, the righthand object's seconds are subtracted from the lefthand object's seconds. Likewise, if the nanoseconds of the righthand object are less than that of the lefthand object, the righthand object's nanoseconds are subtracted from that of the lefthand object. If the righthand object's seconds or nanoseconds are not less than the lefthand's, then the righthand object's seconds or nanoseconds are subtracted from ULONG_MAX, the largest number possible for `unsigned long` integers on the system.

The `aitTimeStamp` object that is assigned the value of the operation is assigned the result values.

Name:

`operator>=()`

Synopsis:

```
#include <aitTypes.h>
int operator>=(const aitTimeStamp &lhs,
               const aitTimeStamp &rhs);
```

Destruction:

The `operator>=()` function is invoked whenever two `aitTimeStamp` operators are compared using the >= operator. If the seconds from the lefthand `aitTimeStamp` object are greater than the righthand object's seconds, True (1) is returned. If the seconds are equal and the nanoseconds from the lefthand `aitTimeStamp` object are greater than or equal to the righthand object's nanoseconds, True (1) is also returned. Otherwise, False (0) is returned.

Name:

`get()`

Synopsis:

```
#include <aitTypes.h>
void get(unsigned long &tv_secOut, unsigned long
&tv_nsecOut);
```

Description:

The `get()` function can be used to retrieve the seconds and nanoseconds of an `aitTimeStamp` object. The function accepts two arguments by reference, so both arguments must be lvalues. The seconds are written into the first lvalue and the nanoseconds into the second lvalue.

Name:

`operator double(), operator float()`

Synopsis:

```
#include <aitTypes.h>
operator float();
operator double();
```

Description:

The `operator float()` and `operator double()` functions return a value of `float` or `double` when an `aitTimeStamp` object is typecast as `float` or `double` as in,

```
   double Seconds;
   Seconds = (double)aitTimeStampObject;
```

The value returned will be a floating-point number the timestamp in seconds.

Name:

```
getCurrent()
```

Synopsis:

```
#include <aitTypes.h>
static aitTimeStamp getCurrent();
```

Description:

`getCurrent()` is a public, static member function that returns an `aitTimeStamp` object with the current time.

CLASS: osiTime

Declared: osiTime.h

The `osiTime` class or operating system-independent time class keeps track of a timestamp. It is used by the server library in conjunction with the `osiTimer` class. It has two members of type `unsigned long` that represent seconds and nanoseconds. These members can be written and retrieved separately, or they can be written and retrieved combined as one floating-point number. By overloading the +, -, >, <, >=, and <= operators, the class allows for `osiTime` objects to be compared to each other, added to each other, or subtracted from each other.

Destruction:

The `osiTime` class has no destructor because it needs no pre-destruction cleanup. If an application derives a class from `osiTime`, it is reponsible for providing any necessary cleanup for that class.

Public Member Functions:

```
      osiTime ();
      osiTime (const osiTime &t);
      osiTime (const unsigned long secIn,
               const unsigned long nSecIn);
      osiTime (double t);
      osiTime operator+= (const osiTime &rhs);
      osiTime operator-= (const osiTime &rhs);
      static friend osiTime operator+ (const osiTime &lhs,
                            const osiTime &rhs);
      static friend osiTime operator- (const osiTime &lhs,
                       const osiTime &rhs);
static friend int operator>= (const osiTime &lhs,
                     const osiTime &rhs);
static friend int operator>(const osiTime &lhs,
                     const osiTime &rhs);
```

```
static friend int operator<= (const osiTime &lhs,
                     const osiTime &rhs);
static friend int operator<(const osiTime &lhs,
                     const osiTime &rhs);
void get(long &secOut, long &nSecOut) const;
void get(unsigned long &secOut, unsigned long &nSecOut);
void get(unsigned &secOut, unsigned &nSecOut) const;
void get(unsigned long &secOut, long &nSecOut) const;
operator double() const; operator float() const;
void show(unsigned);
static osiTime getCurrent();
```

Name:

`osiTime()`

Synopsis:

```
#include <osiTime.h>
osiTime ();
osiTime (const osiTime &t);
osiTime (const unsigned long secIn,
         const unsigned long nSecIn);
osiTime (double t);
```

Description:

`osiTime()` is the class constructor. It's overloaded to provide several means to initialize the class, i.e., initialize the seconds and nanoseconds of the class. The first constructor, osiTime(), accepts no arguments and initializes both the seconds and nanoseconds to zero. The second constructor, `osiTime(const osiTime &t)`, accepts an `osiTime` object, `t`, by reference and initializes the object for the constructor was called with the seconds and nanoseconds of `t`.

The third constructor accepts two `unsigned long` integers, `secIn` and `nsecIn`, as its arguments and initializes the seconds to `secIn` and the nanoseconds to `nsecIn`.

The fourth constructor accepts a value of type `double`. It initializes the seconds and nanoseconds using this value. The seconds are initialized to the integral part of the double value while the nanoseconds are initialized to the fractional part.

Name

`operator+=()`

Synopsis:

```
#include <osiTime.h>
osiTime operator+= (const osiTime &rhs);
```

Description:

The `operator+=()` function is a public member function. It's invoked when the two `osiTime` objects appear on both sides of a += sign in an expression. The seconds of the object on the righthand side are added to the seconds of the object on the lefthand side. The nanoseconds of the object on the lefthand side are also added to the nanoseconds of the object on the lefthand side. Thus, the seconds/nanoseconds of the object on the lefthand side are incremented by the seconds/nanoseconds of the object on the righthand side.

Name:

```
operator-=()
```

Synopsis:

```
#include <osiTime.h>
osiTime operator-= (const osiTime &rhs);
```

Description:

The `operator-=()` function is a public member function that is invoked when two `osiTime` objects appear on either side of the -= operator in an expression. The seconds and nanoseconds of the righthand object are subtracted from the seconds and nanoseconds of the lefthand object. However, if the seconds from the lefthand object are less than the seconds from the righthand object, the seconds from the righthand object are subtracted from ULONG_MAX. And if the nanoseconds from the lefthand object are less than the those from the righthand object, the nanoseconds from the righthand object are subtracted from the number of nanoseconds in a second, and then the nanoseconds from the lefthand object are added to the result. The seconds member is then decremented.

The results of the -= operation are of course assigned to the object on the lefthand side of the operator.

Name:

```
operator+()
```

Synopsis:

```
#include <osiTime.h>
static friend osiTime operator+ (const osiTime &lhs,
                        const osiTime &rhs);
```

Description:

The `operator+()` function is a friend function that is invoked when two `osiTime` objects appear on either side of the + operator in an assignment expression as in,

```
Object1 = Object2 + Object3;
```

The seconds from `Object2` are added to `Object3`; the nanoseconds from `Object2` are also added to `Object3`. The resulting values are written into the object on the lefthand side of the assignment operator, `Object1` in this case.

Name:

`operator-()`

Synopsis:

```
#include <osiTime.h>
static friend osiTime operator- (const osiTime &lhs,
                        const osiTime &rhs);
```

Description:

The `operator-()` function is a friend function that is invoked when two `osiTime` objects appear on either side of the - operator in an assignment expression as in,

```
Object1 = Object2 - Object3;
```

If the seconds from `Object2` are less than the seconds of `Object3`, the seconds from `Object3` are subtracted from `ULONG_MAX`, the largest number possible for an unsigned integer on the system. Otherwise, the seconds of `Object3` are subtracted from the seconds of `Object2`. If the nanoseconds from `Object2` are less than those of `Object3`, then one second is borrowed from the seconds value, the nanoseconds of `Object3` are subtracted from that second, and the nanoseconds of `Object2` added to those nanoseconds. Otherwise, the nanoseconds of `Object3` are subtracted from those of `Object2`.

The resulting values are written into the object on the lefthand side of the assigment operator, `Object1` in this case.

Name:

`operator>=()`

Synopsis:

```
#include <osiTime.h>
static friend int operator>= (const osiTime &lhs,
                         const osiTime &rhs);
```

Description:

The `operator>=()` function is invoked when two `osiTime` objects appear on either side of the >= operator. The seconds/nanoseconds of the lefthand object are compared with seconds/nanoseconds of the righthand object. If the the seconds/nanoseconds of the lefthand object are indeed greater than or equal to those of the righthand object, True (1) is returned. Otherwise, False (0) is returned.

Name:

`operator>()`

Synopsis:

```
#include <osiTime.h>
static friend int operator>(const osiTime &lhs,
                         const osiTime &rhs);
```

Description:

The `operator>()` function is a static friend function that is invoked when two `osiTime` objects appear on either side of the `>` operator. The seconds/nanoseconds of the lefthand object are compared with seconds/nanoseconds of the righthand object. If the the seconds/nanoseconds of the lefthand object are indeed greater than those of the righthand object, True (1) is returned. Otherwise, False (0) is returned.

Name:

`operator<=()`

Synopsis:

```
#include <osiTime.h>
```

```
static friend int operator<= (const osiTime &lhs,
                     const osiTime &rhs);
```

Description:

The `operator<=()` function is a static friend function that is invoked when two `osiTime` objects appear on either side of the <= operator. The seconds/nanoseconds of the lefthand object are compared with the seconds/nanoseconds of the righthand object. If the seconds/nanoseconds of the lefthand object are indeed less than or equal to those of the righthand object, True (1) is returned. Otherwise, False (0) is returned.

Name:

`operator< ()`

Synopsis:

```
#include <osiTime.h>
static friend int operator<(const osiTime &lhs,
                     const osiTime &rhs);
```

Description:

The `operator<()` function is a static friend function that is invoked when two `osiTime` objects appear on either side of the < operator. The seconds/nanoseconds of the lefthand object are compared with the seconds/nanoseconds of the righthand object. If the seconds/nanoseconds of the lefthand object are indeed less than those of the righthand object, True (1) is returned. Otherwise, False (0) is returned.

Name:

`get()`

Synopsis:

```
#include <osiTime.h>
void get(long &secOut, long &nSecOut) const;
void get(unsigned long &secOut, unsigned long &nSecOut)
const;
void get(unsigned &secOut, unsigned &nSecOut) const;
void get(unsigned long &secOut, long &nSecOut) const;
```

Description:

`get()` can be used retrieve the seconds and nanoseconds from the `osiTime` object. All forms accept two integer-type arguments, `secOut` and `nSecOut`, which are passed by reference; therfore, the arguments passed to them should be lvalues. The object's seconds will be written into `secOut` and nanoseconds into `nSecOut`. The variations for `get()` are simply provided to allow for different types to be retrieved.

Name:

`operator double(), operator float()`

Synopsis:

```
#include <osiTime.h>
operator double() const;
operator float() const;
```

Description:

The `operator double()` and `operator float()` functions are invoked whenver an osiTime object is typecast using (`double`) or (`float`). These functions return floating-point value that represents seconds and nanoseconds of the object, where the seconds determine the integral part and the nanoseconds the fractional part. Of course, the `operator double()` function returns a value of type `double`, and the `operator float()` function returns a value of type `float`.

Name:

`show()`

Synopsis:

```
#include <osiTime.h>
void show(unsigned);
```

Description:

`show()` displays the seconds and nanoseconds of the `osiTime` object.

Name:

```
getCurrent()
```

Synopsis:

```
#include <osiTime.h>
static osiTime getCurrent();
```

Description:

`getCurrent()` is a static, public member function that returns an `osiTime` object that is stamped with the current time.

CLASS: osiTimer

Declared: osiTimer.h

The `osiTimer` provides applications with a way to sleep or delay a task for a specified time. After the specified time expires, the member function `expire()` is called. `expire()` is a pure virtual function that an application will need to redefine in a derived class. It can be redefined to perform whatever task is needed upon expiration, i.e., after the specified time has expired.

Before `expire()` is called, the virtual function `again()` is called. `again()` is called to determine if the `osiTimer` object should sleep for another delay period. `again()` returns an enumerated value, `osiBool`, whose enumerators are `osiTrue` and `osiFalse`. If `again()` returns `osiTrue`, the `osiTime` object will call `delay()`, a virtual function that by default returns an `osiTime` object of one second but which can be redefined to return an `osiTime` object that has longer time period. After `again()` returns `osiTrue` and `delay()` returns, the `osiTime` object sleeps for the amount of seconds specified by the `osiTime` object returned by the last call to `delay()`. If `again()` returns `osiFalse`, `destroy()` is called. `destroy()` is also a virtual function which by default deletes the this pointer using the `delete` operator.

Thus, the algorithm for an `osiTimer` or derived-class object is,

A. Initialize `osiTimer` base class with initial delay time.
B. After time expires, `again()` is called.
C. If `again()` returns `osiTrue`, call `delay()`, then `expire()`, then sleep for the amount of time returned by delay.
D. If `again()` returns `osiFalse`, `expire()` is first called, after which `destroy()` is called.

Destruction:

When `again()` returns `osiFalse`, `expire()` is called and then `destroy()` is called to delete the object. `destroy()` is a virtual function that by default deletes the `this` pointer using the `delete` operator. Thus, `destroy()` will only work if the derived class object is created using the `new` operator. `destroy()` can be redefined so that it duly destroys the `osiTime` or derived-class function.

The `osiTimer` class has a virtual destructor, `~osiTimer()`, that will perform any necessary cleanup for the `osiTime` base class and its base classes. A virtual destructor guarantees a certain calling order in the destructors of derived classes. If the destructor of a base class is virtual, then when a derived class object is deleted, the derived class' destructor is called, after which the base class destructor is called.

Thus, the application is responsible for providing for any necessary cleanup for any class derived from `osiTimer`.

Public Member Functions:

Non-Virtual

```
osiTimer (const osiTime &delay);
```

Virtual

```
virtual ~osiTimer();
virtual void expire()=0;
virtual void destroy();
virtual osiBool again();
virtual const osiTime delay();
virtual void show (unsigned level);
virtual const char *name();
```

Name:

```
osiTimer()
```

Synopsis:

```
#include <osiTimer.h>
osiTimer (const osiTime &delay);
```

Description:

`osiTimer()` is the class constructor. It accepts on argument by reference: an `osiTime` object. The `osiTimer` object will use the seconds specified in the `osiTime` object as the initial delay period. If `again()` returns `osiTrue` after the initial delay period, then the number of seconds used for the subsequent delay are returned by the `delay()` virtual function, which returns an `osiTime` object specifying a number of seconds. Therefore, the `osiTime` object passed to the constructor is only used for the initial delay period. That returned by `delay()` is used for all subsequent delay period.

Name:

```
~osiTimer()
```

Synopsis:

```
#include <osiTimer.h>
virtual ~osiTimer();
```

Description:

`~osiTimer()` is the class destructor. It is a virtual destructor that provides the necessary cleanup for the `osiTimer` base class. It also guarantees that when a derived-class object is destroyed, the destructors in the hierarchy are called in a certain order where the last derived class destructor is called first and the base class destructor is called last.

Name:

```
expire()
```

Synopsis:

```
#include <osiTimer.h>
virtual void expire()=0;
```

Description:

`expire()` is a pure virtual function. It is called after the delay period expires. It can be redefined to perform whatever task is needed. It is called after `again()` and `delay()`.

Name:

`destroy()`

Synopsis:

```
#include <osiTimer.h>
virtual void destroy();
```

Description:

`destroy()` is called when the `osiTimer` or derived-class object is no longer needed, that is, after `again()` returns `osiFalse` and after `expire()` is called. It is meant to duly cause the `osiTimer` object to be destroyed. It is a virtual function that by default deletes the `this` pointer using the `delete` operator. Thus, the default version will only work if the object was created using the `new` operator.

Name:

`again()`

Synopsis:

```
#include <osiTimer.h>
virtual osiBool again();
```

Description:

The internals of the `osiTimer` base class call `again()` to determine if the cycle should repeat, that is, if after calling `expire()`, the object should arm the delay timer again. `again()` returns either `osiTrue` or `osiFalse`. If it returns `osiTrue`, `delay()` will be called to determine the number of seconds of the delay, `expire()` will be called, and then the object will sleep for the specified delay period again. If it returns `osiFalse`, `expire()` and then `destroy()` are called.

`again()` is a virtual function that by default returns `osiFalse`.

Name:

`delay()`

Synopsis:

```
#include <osiTimer.h>
virtual const osiTime delay();
```

Description:

`delay()` is a virtual function called by the base class internals after `again()` returns `osiTrue`. By default, it returns an `osiTime` object of one second. An application can redefine `delay()` to return an `osiTime` object that specifies a longer or shorter time period.

Name:

`show()`

Synopsis:

```
#include <osiTimer.h>
virtual void show (unsigned level);
```

Description:

`show()` is a virtual function that can be called to display the current state of the `osiTimer` object. It's a virtual function that by default displays the amount of time remaining in the delay, whether `again()` returns `osiTrue` or `osiFalse`, the name of the derived class, and the state of the object. It can be redefined to display whatever.

It accepts an unsigned integer that represents the debug level.

Name:

`name()`

Synopsis:

```
#include <osiTimer.h>
virtual const char *name();
```

Description:

`name()` is called by the default version of `show()` but can be called by the application. It's a virtual function that should display the name of the class derived from the `osiTime` class. By default it returns the string "unknown class deriving from `osiTimer`."

CLASS: gddAppFuncTable<PV>

Declared: gddAppFuncTable.h

The `gddAppFuncTable<PV>` template class allows a server tool to install a number of "read" functions in a function table. When the functions are installed, a server tool specifies a corresponding application type for the function. Then when a read operation is requested, the server tool can then call the `gddAppFuncTable::read()`. As one of its arguments, `read()` accepts a `gdd` object. This `gdd` object can be a `gddContainer` object, in which case it contains other `gdd` objects, or it can be a single `gdd` object such as a `gddScalar` object. If a `gddContainer` object, `read()` will step through the object. For each `gdd` object within the container type, `read()` will access its application type and call the function that was installed for its application type. If the object passed to `read()` isn't a `gddContainer` object, `read()` will simply call the function installed for its application type.

If no function is found for a particular application type, that is, no function has been installed for that application type, an error message will be printed and the next `gdd` object in the container will be accessed.

Each function to be installed must return a `gddAppFuncTableStatus` code, which indicates the error or success of the read function. In addition, the function accepts only one argument, a `gdd` object. Thus, a basic format is

```
gddAppFuncTableStatus casPV::yourFunction(gdd &value);
```

where `casPV` is the class which the function is a member of. The function must be a member of a class derived from the `casPV` class and `yourFunction()` is the name of the function. It can, of course, have any name, as long as the format is the same.

To understand how the `gddAppFuncTable` operates, it's necessary to understand what an application type is. An application type is simply an unsigned integer (`aitUint16`) that is part of a series of unsigned integers that represent a code. In this code, an `unsigned integer` can be defined to represent an application type. Each application type describes a piece of data and its purpose.

A set of predefined application types is provided for EPICS data. Each of these application types describes a piece of data for EPICS applications. For example, one application type is represented by the constant `gddAppType_status`. The "status" application type describes the data as representing the alarm status for a Process Variable. The application type `gddAppType_value` represents the actual value of the process variable. Other application types can be added to the predefined application types for EPICS applications or even non-EPICS applications.

The existing application types appear in the header file gddApps.h. In addition, some of these application types have corresponding strings such as "status" for the application type `gddAppType_status`.

To use the `gddAppFuncTable<PV>` template, you must declare an instance of the template, specifying the casPV or derived class of the functions. For instance, if a server tool had a class called `ourPV` derived from `casPV` and it wanted to install read functions for that PV class, it would declare a function table in the following manner:

```
gddAppFuncTable<ourPV> ourFuncTable;
```

The most important functions in the `gddAppFuncTable` class-template are `installReadFunc()` and `read()`. None of the other members should be of any concern to the user.

Name:

`gddAppFuncTable()`

Synopsis:

```
#include <gddAppFuncTable.h>
gddAppFuncTable<PV>::gddAppFuncTable(void);
```

Description:

`gddAppFuncTable()` accepts no arguments and thus when the class-template is instanced, the application should not specify any arguments.

Name:

`installReadFunc()`

Synopsis:

```
#include <gddAppFuncTable.h>
gddAppFuncTableStatus
gddAppFuncTable<PV>::installReadFunc(
                        const unsigned type,
                        gddAppReadFunc pMFuncIn);
gddAppFuncTableStatus
gddAppFuncTable<PV>::installReadFunc(
                        const char * const pName,
                        gddAppReadFunc pMFuncIn);
```

Description:

An application uses `installReadFunc()` to install functions in the application table that it has created. There are two forms of `installReadFunc()`. One accepts a character string for the application type as its first argument and the name of the function as its second argument. The other accepts a constant for the application type as its first argument and the name of a function as its second argument.

Here is an example of the call to `installReadFunc()` where the application type is a represented by a character string and where `ourFuncTable` is the name of the function table:

```
ourFuncTable.installReadFunc("status", readStatus);
```

Here, the application type is the "status" application, a type predefined for EPICS applications. The function `readStatus()` is being installed to read the status of a PV's alarm.

And here is an example of the call to `installReadFunc()` where the application type is a constant defined in gddApps.h:

```
ourFuncTable.installReadFunc(gddAppType_status,
readStatus);
```

Here, the application type and function name are the same except that the application type is an unsigned integer constant defined in gddApps.h.

Note that if the application type is passed as a character string, the string must have already been defined to represent a particular application type, that is, they must have been installed in a table of application types.

Name:

`read()`

Synopsis:

```
#include <gddAppFuncTable.h>
 gddAppFuncTableStatus gddAppFuncTable<PV>::read(PV
&pv,
                        gdd &value);
```

Description:

An application calls read when it wants to read a `gdd` object's value or values. Whether a `gddContainer` object or any other `gdd` object, by calling `read()` and passing it the `casPV` or derived-class object for the PV that is being read, and the `gdd` object passed to the `casPV::read()` function. `read()` then calls the appropriate function to read the appropriate value(s) from the `gdd` object(s).

For instance, suppose a server tool defined all the functions required to read all predefined EPICS application types and that it installed these in the application table `myFuncTable`. Suppose these functions are members of the class `myPV`, which is derived from the `casPV` class. When `myPV::read()` is called, it passes itself, that is, the `myPV` object for which `read()` was called, and the `gdd` object passed to `myPV::read()` to `gddAppFuncTable::read()`. Provided that all the appropriate functions have been installed, `gddAppFuncTable::read()` will call all the necessary read functions for the `gdd` object.