Channel Access Portable Server

Application Interface (API) Tutorial

Philip Stanley

August 20, 1999

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.

Chapter 1: Introduction

1. Heretofore Organization of EPICS Software

The basic organization of the EPICS control system software can be divided into three parts:

1. The client-side tool set which consists of *client applications* like EDD/DM, the alarm manager tool, and the archiver tool.
2. The Channel Access communications software, through which client applications can communicate with remote databases.
3. The IOC database software which includes read and write routines, record support modules, device-support routines, scanners, and monitors, among other components.

The client applications provide the interface to the system, Channel Access provides client applications with a means to communicate with the database, and the database software implements the control algorithms needed to control and monitor devices.

Channel Access is the   gateway   between the client applications and the database software. In its entirety, Channel Access provides a way for applications to establish network-transparent connections to *process variables* (PVs) that may be running on remotely-located hosts, usually on the same LAN/WAN. In EPICS, a process variable refers to a record or a record's field. Once a connection has been established, the client can, via Channel Access, change the process variable's value, read its value, or establish monitors on the process variable, which instruct Channel Access to notify the client application when the PV's value changes.

Channel Access is a client/server application. In a typical situation, the client would run on a workstation and the server on an IOC. The server is loaded onto the IOC when the database is initialized. The client is created when any client-side program calls any of the requests from the Channel Access *Client Library*. This library consists of a number of simplified function calls that a client application can use to establish connections or *channels* to process variables and then read from or write to them. Because the Client Library is a simplified interface to communicating via channels, the programmer who is developing a client application does not need to be concerned with any of the complexities or internals involved in establishing network connections or sending data to and reading data from those channels. For instance, to establish a connection, a programmer simply has to call the following function,

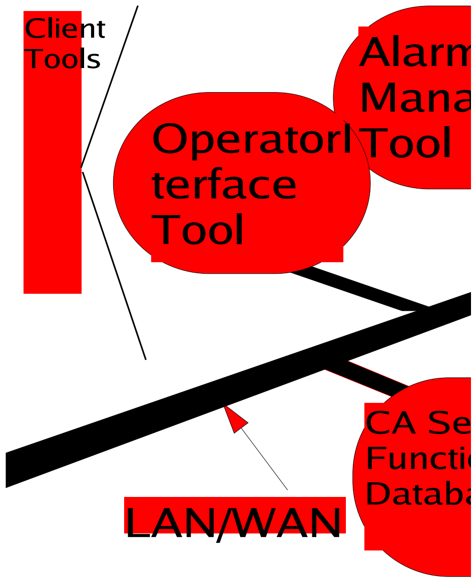
```
ca_search(Channel_name, Channel_ID);
```

and provide a channel ID and the name of a process variable; the programmer doesn't have to bother with TCP/IP addresses and port numbers and all such details.

The client's requests are sent to the server, and the server attempts to fulfill each request. Actually, the server merely calls the database routines to perform all read, write, or monitor actions. So, for instance, when a client requests a process variable's value, the server receives the request and calls the database routine `dbGet()` or one of its aliases. The database routine then accesses the record and returns the value to the server which returns it to the client. Thus, all Channel Access client requests that read data, write data, or monitors data changes are mapped to specific EPICS database routines, and thus Channel Access's functionality is tied to the EPICS IOC database software, though well-isolated from changes to it.

2. Limitations Imposed by this Organization

Figure 1-1: Heretofore EPICS Organization.



The above approach has served EPICS users and developers well in the past, but there are some limitations and problems that have accompanied it.

Limited Use of Client Tools

Heretofore, such useful tools as the Display Manager and the Archiver could only be used with the EPICS function block database. Thus, a DM display could only be used to monitor and control EPICS process variables in an EPICS database. This is because DM, like all EPICS client applications, interfaces with the system by issuing requests from the Channel Access Client Library which sends the requests to the server which maps the requests to EPICS database routines which only operate on EPICS database records and their fields.

There is currently a large number of client tools, including tools that enable an EPICS user to interface the system to such commercial packages as Mathematica, TCL/TK, and IDL. The usefulness and sophistication of client applications written specifically for EPICS such as DM is growing. The generic nature of these interfaces and tools should permit them to be used with non-EPICS applications if they weren't tied to the EPICS database and its IO routines.

Storage of Intermediate Results

Another problem that has accompanied this approach occurs when two client-side tool kits must compute an intermediate result. For instance, if a client application requests the result from an operation that involves several steps or calculations such as an emittance scan, the intermediate results must be stored in a soft record. (*Soft records* refer to EPICS record that read or write data from other records rather than interface to devices.) If another client application requests the same result that involves the same records, the data sent by the two applications could collide producing an unexpected result if mutual exclusion is not built into the algorithm.

Such an approach is also not congruent with the appropriate design and use of client/server software in which the client side should hold most of the burden for storing and converting data, leaving the server to perform more crucial tasks.

3. The Solution: a New Server-Level API for EPICS

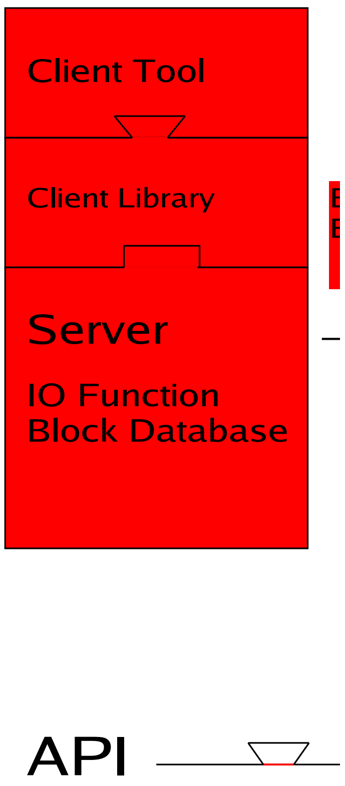
The above limitations could be solved by making low-level changes to the source code. However, this contradicts the tool-based approach of EPICS. In addition, users of EPICS at one site could not benefit from the labor of individuals at another site without changing source code.

The solution for these and other problems has been to develop a new server-level application interface (API) for the Channel Access Server, called the Channel Access Portable Server. Basically, instead of being tied to the I/O routines of the EPICS database, the new server can be tied to non-EPICS I/O routines, thus allowing any EPICS client applications to interface with those applications. Another use for the server would be to create a server dedicated to a specific low-priority task. In addition, the Portable Server can be used to store intermediate data used by one client application without the danger of collision with another client.

The Portable Server consists of a C++ library with a simple class interface. Using this interface, a developer can create a Channel Access server that can interface with the EPICS database as well as other applications. By making the server another module in the EPICS tool set, significant improvements in cost, quality, and distribution can be made. This represents a significant step in the evolution of EPICS.

4. The Server Interface

Figure 1-2: Evolution of EPICS.



This document's purpose is to describe the C++ interface to the Portable Server library. This C++ interface consists of nine classes, each of which has several member functions that can be used in a server application. Many functions are virtual functions that can be re-defined in a derived class to provide more specific functionality. A few of these functions are *pure virtual* functions, which means that a server application *must* provide implementations for them. Nonetheless, the amount of code needed to implement a basic server tool is minimal, and most of the complexity of Channel Access is hidden beneath the interface using the object-oriented approach.

The following nine classes comprise the Portable Server API:

- the server class, `caServer`
- the process variable class, `casPV`
- `pvExistReturn`
- `pvCreateReturn`
- the channel class, `casChannel`

in addition to the asynchronous IO classes:

- `casAsyncPVExistIO`
- `casAsyncCreatePVIO`
- `casAsyncReadIO`
- `casAsyncWriteIO.`

Only the first four classes listed above are required to implement a functional server tool. The channel class and the asynchronous IO classes can be used to add more functionality, but are not a necessary part of a server application.

This document describes the C++ API and not the internal workings of the server library which are very complicated. The programmer does not have to be an expert in C++ in order to understand how the interface works or even how to write a basic server tool. However, some familiarity with virtual functions and class inheritance is necessary.

Because of the wide range of possible applications of the Portable Server, the examples in this document cannot show all its possible uses or the possible implementations for the virtual functions, but can only explain how to use the functions in its library.

Chapter 2: Getting Started

1. Basic Concepts

The Server Library and Server Tools

The *server library* refers to the software that lies beneath the C++ class-interface to the Portable Server. The developer only needs to know how to use the interface in order to create a *server tool*. A server tool is any specific application written by a developer using this interface. A user should not be concerned with the server library because it has been intentionally hidden from the user and deals with things like the TCP/IP and UDP protocols, sending beacons to the client, and other ❽nternals❾which the developer need not know.

The server library interacts with the server tool by calling the functions that form the C++ interface, the functions that the server tool provides. For instance, when a client requests to write a value to the server, the server library receives the request and calls the `write()` function, which is part of the C++ interface and a member of the `casPV` class. The server tool provides its own specific implementation of the `write()` function or uses the default implementation provided in the base class. Thus, the server tool's `write()` function will be called to change a process variable's value when the client requests to change the value.

PVs

The term *Process Variable* or PV is a general term used throughout EPICS and other control system applications. It basically refers to any changeable piece of data that, for example, controls a device. For instance, speaking of the EPICS database software, a stepper motor record that controls a stepper motor is a PV: changing the record's value changes the behavior of the stepper motor.

As far as the Portable Server is concerned, a PV is a variable which the server tool keeps track of. The server tool provides the client with the current value of the PV when the client requests an update (a read operation). The server tool also changes the current value of the PV when the client requests that the value be changed (a write operation). The server tool can also inform the client when the PV's value has changed if a client requests that it be informed (monitoring). When the client makes a request, the server library receives it and calls the appropriate function provided by the server tool.

A *channel* differs from a PV in that a channel is a connection between a particular client and a PV. For instance, on a particular server tool there is one PV called Process Variable A. If client X and client Y each establish a connection to Process Variable A, then two channels are created for Process Variable A, one channel for client X and one for client Y. Potentially, both clients have access to read from the PV and write to the PV; however, there can be different levels of access for each channel. For instance, client X can have both read and write access to PV A, while client Y has read access only.

A PV can have several *attributes*. For example, suppose that there is an analog PV. In addition to the actual analog value, an analog PV can have alarm limits, the alarm status, the high operating range value, the low operating range value, and so on. Of course, these are attributes from an EPICS PV. Generally, though, PVs from other applications will also have attributes. The server tool will also want to keep track of a PVs attributes. In fact, the term PV can also refer to the actual value of the PV in addition to its attributes.

Data Types and the gdd Library

Channel Access clients make requests to read or write data using database request types or `DBR` types. These types can be found in the `db_access.h` header file. A client can read data using *simple* data types like `DBR_CHAR` or `DBR_INT`, and it can also read data using *compound* data types like `DBR_STS_CHAR` or `DBR_GR_INT`. The simple data types are used by clients to read, write a single scalar value or a single array from, to a database; for instance, `DBR_FLOAT` can be used to read the floating-point value of an analog channel. A compound type such as `DBR_STS_FLOAT` can be used to read the value in addition to some of the PV's attributes. The server would then return a a structure called `dbr_sts_float` which had three members: the value of the PV, its alarm status, and its severity.

A client can request a data type that is different from the native type of the PV. In an EPICS database, the native types are of type `DBF_FLOAT`, `DBF_STRING`, or one of the other DBF types. For instance, if an analog PV were of type `DBF_FLOAT` and a client requested to read it using the `DBR_STRING` request type, the Channel Access server performs any conversions on the data before sending it back to the client; that is, it converts the floating-point value to a character string representing the value.

The new Portable Channel Access Server is no longer tied to dealing only with `DBR` and `DBF` types, though it is designed to be compatible with existing EPICS clients. Instead of consisting of `DBR` or `DBF` types, the data in the Portable Server is described by two types:

1. The *architecture-independent type*. When EPICS clients communicate with a server, the client is often running on a host with a different architecture than the server. There must be a way for the client and the server to exchange data that is independent of the architecture on each machine. The use of architecture-independent types enables the client and the server to exchange data even though they are running on hosts which represent data differently, for example, when the client's host represents an `int` object as 16 bits and the server's host represents an `int` object as 32 bits. An example of an architecture-independent type is the type `aitUint32`, an unsigned integer-type of 32 bits, which on a Sun Sparc workstation is defined as `unsigned int`,
   ```
        typedef unsigned int aitUint32;
   ```
   but could be defined as `unsigned long` on a machine where type `int` is 16 bits and type `long` is 32 bits. The architecture-independent types can be found in `aitTypes.h`.
2. The *application type*. The old CA Server also used architecture-independent types. Application types are, on the other hand, a new way of dealing with data used by the Portable CA Server and the `gdd` library. An application type is a way of describing how a particular piece of data is used. For instance, there's a predefined application type called ❽precision❾ This application type describes an attribute of a floating-point PV that determines the decimal precision of the PV's value. The ❽ value❾application type describes data that represents the current state of the PV, i.e., its actual value.

The header file `gddAppTable.h` shows some of the pre-defined application types such as value, limits, and status. These predefined application types enable a Portable Server to communicate with existing EPICS client applications. New application types can be created, however, to describe new uses of data. Thus, the use of application types creates an ❽open-ended❾way for clients to communicate with the server. Channel Access client applications are taken care of by mapping certain `DBR` types to application types. For example, all simple or atomic data types like `DBR_FLOAT` can be described by the ❽ value❾application type since they simply retrieve a single value. Other application types have been created to satisfy requests for compound types such as `DBR_STS_FLOAT`, which itself is just a container for other application types, namely the ❽status,❾❽severity, ❾and ❽value❾application types. An application type can therefore be a container for other application types.

A Portable Server tool doesn't have to know how to create application types, but it must know how to satisfy requests of any application types with which it may have to deal with. For instance, if a client defines a new application type called ❽foo,❾then the

Portable Server must know what to do when it receives a request to read a piece of data that is described by the application type ❽foo.❾

The Portable CA Server uses the `gdd` library and its associated header files to deal with data. This partially shields the Portable Server from having to deal with the different data types. It simply deals with `gdd` objects which are objects of the `gdd` class or one of its derived classes, `gddAtomic`, `gddScalar`, or `gddContainer`. Taking an object-oriented approach, the `gdd` class and its derived classes contain members for the actual data and functions to manipulate the data. The most important member of a `gdd` class is the `data` member, which should not be accessed directly. The `data` member is really a union of all the existing architecture-independent types. The `gdd` class also contains members and functions with which to identify the application type of the `gdd` object as well as the current data type of the `data` member. All this information can be accessed through the appropriate member functions of the `gdd` class. The `gdd` class and its derived classes also have members for converting data from one architecture-independent type to another.

The classes derived from `gddÄgddAtomic`, `gddScalar`, and `gddContainerÄ`are used in particular cases. The `gddScalar` class is used to deal with single, scalar values, while the `gddAtomic` class is used to deal with arrays. The `gddContainer` class is used to contain other `gdd` objects.

To write a server tool, a programmer should be familiar with the basics of using the `gdd` library. This document will touch upon a few, but not all, of the basics of the `gdd` library. The `gdd` classes are declared in `gdd.h`. The actual library to be linked with is `libgdd.a`.

Class Basics

As mentioned previously, the programmer's concern is with the C++ interface to the Portable Server library and not the internals of the library itself. The interface has been simplified as much as possible and consists of nine classes:

1. caServer
2. casPV
3. pvExistReturn
4. pvCreateReturn
5. casChannel
6. casAsyncPVExistIO
7. casAsyncCreatePVIO
8. casAsyncReadIO

9. casAsyncWriteIO

A Portable Server tool must use the first four classes, `caServer`, `casPV`, `pvExistReturn`, and `pvCreateReturn`. The other five classes are optionalÄthe asynchronous IO classes such as `casAsyncReadIO` are necessary only for asynchronous IO while `casChannel` is only necessary in order to control access by particular users to a PV. The definitions of all the classes are found in the header file casdef.h.

Though a detailed knowledge of C++ is not necessary to write a server tool, some knowledge of class inheritance and virtual functions is. A brief explanation of inheritance and virtual functions follows.

A class consists of members, which can be either functions, frequently called *methods*, or variables, structures, or other classes. These members can be either *public*, *private*, or *protected* members. When a member is *public*, it can be used in any function, via any object of that class. For instance, if `m` is a public member of class `Y`, `m` can be used by the function `main()` via the object `obj`, even though `main()` is not a member function of `Y`:

```
main()
{
   int i;
   Y obj;
   i = obj.m
}
```

For public members that are functions, this means that they can be called by any other function via an object of that class, even when the calling function has no association with the member's class.

When a member is *protected*, it can only be used by member functions of the member's class and by member functions of any classes derived from the member's class. Thus, in the above example, the last statement `i = obj.m` would be illegal if `m` were a protected member of class `Y`. It would be legal however, if the above statements occurred inside the function `V()` which was a member of class `X`, which was derived from class `Y`, as in the following:

```
void X::V()
{
   int i;
   i = m;
}
```

When a member is *private*, it can be used only by functions that are members of the same class in which it was declared. Thus, the above example would be illegal because `m` is a member of `Y,` and `V()` is a member function of class `X` which is derived from class `Y`. Thus, `m` is accessible only by member functions of class `Y`.

Classes are derived from other classes as in the following declaration where class `X` is derived from class `Y`:

```
class X : public Y {
public:
   int member;
   // . . . other stuff . . .
}
```

The derived class inherits all the members of the base class, the class derived from.

Virtual Functions

A virtual function is a special kind of class member function. Ordinary virtual functions have default implementations in a base class, but can be redefined in a derived class. Using virtual functions is called *dynamic linking* because which function definition to useÄthe function in the base class or the function in the derived classÄis undetermined until compile time. For instance, if a base class `A` has a virtual function called `oopy()` as in the following,

```
class A{
public:
   virtual void oopy()
   {
      printf("Hello.\n");
   }
   // . . .more stuff . . .
}
```

and class `B` derives from class `A`, the user can redefine `oopy()` to do something else:

```
class B : public A{
public:
   void oopy()
   {
      printf("Howdy.\n");
   }
   // ...more stuff...
}
```

If a class derived from class A, class `B` in this instance, redefines the virtual function `oopy()`, then at compile time when `oopy()` is called for a `B` class object, the definition found in class `B` will be linked with at compile time and ❽Howdy❾will be printed when `oopy()` is called at run-time. However, if class B chooses not to redefine `oopy()`, the default implementation of `oopy()` will be linked with at compile time and ❽Hello❾will be printed when `oopy()` is called at run-time for the object. It's important to understand that although a virtual function can be redefined to do a different task, its arguments and its return value cannot be changed. For instance, oopy() in the above example is declared in class `A` as accepting no arguments and returning no value; its redefinition in class `B` cannot then give it an argument of type `int` and make it return a `double` value. If a function with the same name is redefined in a derived class but has different arguments or return value, it's treated as a different function.

Pure Virtual Functions

There are special virtual functions called *pure virtual* functions which don't have default implementations and so *must* be redefined in a derived class. The declaration of pure virtual functions differs slightly from that of a virtual functionÄto the left of it is the assignment operator (=) and a zero, as if zero were being assigned to the function:

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

Most functions in the interface classes are ordinary virtual and not pure virtual functions. There is only one pure virtual functions and it is part of the `casPV` class. Any class with one or more pure virtual functions is called an *abstract* class. An abstract class is one that can only be used as a base class. You can never use an abstract class to create an object, and any attempt to declare an object using an abstract class will fail. Thus, since `casPV` is an abstract class, the following is illegal:

```
main()
{
   casPV pvObject; // COMPILER ERROR!
}
```

As an example of a pure virtual function, suppose that class `Y` is an abstract base class with one pure virtual function and one ordinary virtual function, as in the following declaration:

```
class Y
{
public:
   virtual void func(int arg) = 0;
   virtual void func1(int arg)
   {
      cout <<"arg = "<<arg<<endl
   }
};
```

Since it has a pure virtual function, here class `Y` is an abstract class, so no objects can be declared for it. However, by deriving a class from `Y` and redefining the pure virtual function `func()`, you can create a class which you can use. The following is an example that derives a class `X` from class `Y`, redefines the pure virtual function, but doesn't redefine the other virtual function, `func1()`, as it intends to use the default implementation for the function.

```
class X : public Y
{
public:
   void func(int arg) { cout<<"arg = "<< arg + 1<<endl; }
   // other stuff ...
}
```

Thus, the following code,

```
main()
{
   X Obj;
   int i = 1;
   Obj.func1(i);
   obj.func(i);
}
```

will print

```
arg = 1
arg = 2
```

Destructors

Destructors are functions that provide for any necessary cleanup before a class object is destroyed. They are called automatically before the object's container is destroyed. An object is destroyed either when it goes out of scope, or if the object was created using the `new` operator, when the `delete` operator is applied to the object. Only rarely should a program call a destructor directly. Since the object's container is automatically destroyed, most class members do not have to be explicitly destroyed; only dynamically-allocated members need to be cleaned up prior to the object's destruction. A destructor is declared with the same name as the class preceded by a tilde '~'. For instance, the destructor for class `B` is declared as `~B()`.

A destructor can be declared to be virtual. Virtual destructors guarantee a specific calling order of the class destructors in a derivation hierarchy. If the destructor in a base class is declared virtual, then all the other destructors of any classes derived from the base class are automatically virtual destructors. When an object of a class within the derivation hierarchy is destroyed, its destructor is called in addition to the destructors of all the classes which it is derived from. The order in which the destructors are called is: the base class destructor is called first; then the destructor for the class derived from the base class; then the destructor for the class derived from that class, and so on until the destructor for the class of the object being destroyed is called.

For example, class `First` is a class which has the virtual destructor function `~First()`:

```
class First {
public:
   First(); // Constructor
   virtual ~First(); // Virtual Destructor
   // Other stuff
};
```

If class `Second` is derived from class `First`, and class `Third` from class `Second`, the destructors `~Second()` and `~Third()` will also be virtual destructors, whether declared so or not. If an object is created for class `Third`, when the object is destroyed, the destructors for the classes will be called in the following order: `~First()`, `~Second()`, and `~Third()`.

Most of the classes in the Portable Server interface have virtual destructors. These virtual destructors are defined to clean up any ❽nternals❾of the base class and the server library. What exactly they do shouldn't be of any concern to a programmer, except that they guarantee that the above-described calling order will be invoked whenever an object of a class derived from one of the base classes is destroyed, and thus that the necessary cleanup will occur. The server tool is responsible for providing for any necessary cleanup for classes derived from any of the base classes.

A brief description of the four essential classes of the Server's C++ interfaceÄ`caServer`, `casPV`, `pvCreateReturn`, and `pvExistReturn`Äfollows.

The caServer Class

As mentioned above, each server tool must include a class derived from the `caServer` class. It's two most important functions are the virtual functions, `createPV()` and `pvExistTest()`, which a server tool must define in a derived class.

The `caServer` class determines the basic characteristics of the server tool such as how many simultaneous IO operations are allowed, the maximum allowable length of a PV name, the debug level, that is, the amount of information printed about the current status of the server, and so on. The most important tasks of the `caServer` class are to inform the server library (which will inform the client) if a PV "is associated with" the server tool and to create a PV when a client wishes to establish a connection to a PV, given that the PV is the server tool's responsibility.

These tasks are performed by the virtual functions `pvExistTest()` and `createPV()`. Since neither has a default implementation that really does anything, the server tool must provide an implementation for each. The server library calls `pvExistTest()` to determine if a PV exists. It does this when a client has issued a search request for a PV. The server library receives this request, and calls `pvExistTest()`, which has as one of its arguments a character string representing the PV name. The server tool should perform whatever algorithm necessary to determine if the PV does exist (meaning, that the server tool is "responsible" for the specified PV) and return an object of the `pvExistReturn` class which is a container for an enumerated type, `pvExistReturnEnum`, whose enumerations are `pverExistsHere`, `pverDoesNotExistHere`, and `pverAsyncCompletion`. Actually, the server tool doesn't have to directly initialize a `pvExistReturn` class, but can instead simply return one of the enumerations. It should return `pverExistsHere` to indicate that the PV exists, `pverDoesNotExistHere` to indicate that it doesn't exist, and `pverAsyncCompletion` to indicate that the server tool will determine the PV's existence at a later time.

After the PV has been found to exist, the server library will call `createPV()`. The return value of `createPV()` is a `pvCreateReturn` object. However, as with `pvExistTest()`, the server tool doesn't have to explicitly initialize a `pvCreateReturn` object. Instead, it can simply return the following: a pointer to a `casPV` object (this includes an object of a class derived from `casPV`); a `casPV` object; the status code `S_casApp_pvNotFound`, `S_casApp_noMemory`, or `S_casApp_asyncCompletion`. The first two return values will indicate a successful operation whose status code is `S_casApp_success`, unless the pointer is a NULL pointer, in which case the status of the operation will be `S_casApp_pvNotFound`. The status code `S_casApp_pvNotFound` tells the server that the PV no longer exists in this server tool, the status code `S_casApp_noMemory` indicates an error allocating memory for a `casPV` object, and the status code `S_casApp_asyncCompletion` indicates that the server tool will perform the creation task at a later time.

The general idea is that the server tool will redefine `createPV()` to create a `casPV` object; however, the server tool can also precreate a series of `casPV` objects for all the PVs it is responsible for, and then return a pointer or reference to them in `createPV()`. As one of its arguments, `createPV()` is passed the name of the PV as a character string.

There are other virtual functions in the `caServer` class which server tool may want to provide definitions for, but redefining `pvExistTest()` and `createPV()` is essential for a server tool. The default definition of `pvExistTest()` returns `pverDoesNotExistHere`, and the default version of `createPV()` returns `S_casApp_pvNotFound`. Thus, regarding the `caServer` class, the programmer's main responsibility will be to derive a class from it and provide implementations of the `pvExistTest()` and `createPV()` functions.

The casPV Class

The `casPV` class is used by the server library to interact with a particular PV. For example, when a client issues a request to read a PV's value, it will issue the request, the server library will receive the request, and call `read()`, which is a virtual function of the `casPV` class. The function `read()` is designed so that a server tool can redefine it to write the PV's current value into the gdd object that is passed by reference as one of its arguments. The server library will then return this value back to the client. A similar thing occurs when a client wants to write a value to a PV: it issues a request, the server library receives the request and calls `write()`, which is a member of the `casPV` class and is also a virtual function. `write()` is designed so that a server tool can redefine it to retrieve the value from the `gdd` object passed as on of its arguments and change the current value of the PV to match the new value. Both `read()` and `write()` should return a status code indicating the success/failure of the operation. These status codes are defined in casdef.h, and have the form S_casApp_*nnn*, such as `S_casApp_succcess`. These status codes were created specifically for the interaction between the server tool and the server library.

Thus, the programmers main task in regards to the `casPV` class will be to derive a class from `casPV` and redefine the virtual functions `read()` and `write()` in it.

It should be noted that the default versions of `read()` and `write()` simply return a status code indicating that the operation isn't supported, namely, `S_casApp_noSupport`. It should also be noted that `read()` and `write()` will be called for a PV.

Other important functions of the `casPV` class are the `beginTransaction()` and `endTransaction()` functions. `beginTransaction()` is called before either `read()` or `write()` is called; and `endTransaction()` is called after either `read()` or `write()` is called. There are other important functions in the `casPV` class such as the `interestRegister()` function which is called when a client requests a monitor on a PV and `interestDelete()` which is called when a client requests that a monitor on a PV be terminated. As with most of the other member functions of the `casPV` class, the above functions are virtual functions that have default implementations, but these implementations perform no real task and are provided so that a server tool can create `casPV` objects and ignore those functions which it doesn't need.

2. A Simple Server Tool

This section presents an example of a simple, trivial server tool called `myServer`. It derives classes from the `casPV` and `caServer` class, and only deals indirectly with the `pvCreateReturn` and `pvExistReturn` classes. These are the minimum classes a server tool must include to be functional.

The full listing for this program can be found in Appendix A, which also explains how to compile server tools, i.e., what libraries to link with and in what order. Examples using the channel and asynchronous I/O classes will be presented in a subsequent chapter when access rights and asynchronous I/O are discussed. Though trivial, this program should demonstrate the basics of Portable Server API and also the basics of using the `gdd` library.

The Classes

Four separate source files comprise our program. Each of them contains relatively little code.

- `myServer.h`ÄContains the class declarations.
- `myPV.cc`ÄContains the function definitions for the `myPV` class, a class derived from `casPV`.
- `myServer.cc`ÄContains the function definitions for the `myServer` class, the class derived from `caServer`.
- `Server.cc`ÄContains `main()`, the actual program to be run.

First, let's examine the classes in myServer.h. There are three:

- `pvAttr`, a class which is not part of the Portable Server interface but which is used here to keep track of the attributes associated with each PV including its value.
- `myServer`, which is derived from `caServer`.
- `myPV`, which is derived from `casPV`.

pvAttr

Each PV in our server will have a `pvAttr` object associated with it. Its members keep track of the PV's value and its attributes, and its functions can be used to access them.

Here is the class declaration for `pvAttr`. Note that all the actual members are private and the functions for accessing them are public. The program must initialize the class by passing to its constructor the name of the PV.

```
class pvAttr {
 public:
  pvAttr (const char *pName) : name(pName)
    {
      pValue = new gddScalar(gddAppType_value, aitEnumFloat64);
      pValue->reference();
         hopr = 100, lopr = 0;
      units = "Jolts";
      high_alarm = 101, high_warning = 95;
     low_warning = 5, low_alarm = -1;
      high_ctrl_limit = 100, low_ctrl_limit = 0;
      precision = 4;
    }

  const aitString &getName () const { return name; }
  double getHopr () const { return this->hopr; }
  double getLopr ()  const { return this->lopr; }
  double getHighAlarm () const { return this->high_alarm; }
  double getHighWarning () const { return this->high_warning; }
  double getLowWarning () const { return this->low_warning; }
  double getLowAlarm () const { return this->low_alarm; }
  double getHighCtrl () const { return this->high_ctrl_limit; }
  double getLowCtrl ()const  { return this->low_ctrl_limit; }
  short getPrec () const { return this->precision; }
  gdd* getVal () { return this->pValue; }
  aitString getUnits () const { return this->units; }
 private:
  const aitString name;
  double hopr;
  double lopr;
  aitString units;
  double high_alarm;
  double high_warning;
  double low_warning;
  double low_alarm;
  double high_ctrl_limit;
  double low_ctrl_limit;
  short precision;
  gdd *pValue;
};
```

Most of the `pvAttr` class declaration should be self-explanatory: a public member function returns the value of a private member. For instance, `getUnits()` returns `units`, an `aitString` object that indicates the units of the PV. These functions will in turn be called by other functions in order to read the sought after value or values. Each of these values is an attribute of the PV; the `high_alarm` member represents the high alarm limit; the `hopr` member represents the high operating range; the `lopr` member, the low operating range, and so on. Of course, these values have nothing to do with a real application like a database, but are simply values that can satisfy a request for a `dbr_ctrl` structure, which is the data type requested by DM clients. Therefore, our program will be able to satisfy DM requests.

The value of the PV itself is stored in the `gdd` object pointed to by the member `pValue`. Remember that there are several types of `gdd` objects which are types created for specific purposes: a `gddScalar` object is meant to hold a single value; a `gddAtomic` object is meant to hold an array, while a `gddContainer` is meant to hold any number of other `gdd` objects. Our server tool will support only the reading and writing of scalar values, for simplicity's sake. Thus, we create a `gddScalar` object pointed to by `pValue`. This is done in `pvAttr()`:

```
      pValue = new gddScalar(gddAppType_value, aitEnumFloat64);
```

There are several ways to initialize a `gdd` object, depending on which type it isÄ`gddScalar`, `gddAtomic`, or `gddContainer`Äand on other factors. For instance, to initialize a `gddAtomic` object, the dimensions of the array can be specified.

Two important characteristics describing a `gddScalar` object are its *application type* and its *primitive type*. An application type describes how a piece of data is used and what it represents. For instance, the application type of a PV's value would be ❽value❾while the application type of a PV's status is ❽status❾ New application types can be created, but a list of predefined ones appear in gddApps.h. The primitive type is simply the architecture-independent type of the data, `aitFloat64`, for instance, which is a floating-point number of 64 bits. The primitive type is actually an enumerated type, the enumerators corresponding to each architecture-independent type. The enumerated type is defined in the aitTypes.h and is called `aitEnum`. We initialize the `gddScalar` object with the constant `gddAppType_value` for the application type and the enumerator `aitEnumFloat64` for the primitive type. Thus, our `gdd` object will be for a floating-point scalar value of 64 bits that will represent the PV's value.

`pvAttr` provides no functions for writing to its members because our server tool doesn't intend to change any of the attributes, only the value of the PV, pointed to by `pValue`. Instead of using a separate function for accessing and changing this value, the PV's value will be written to directly by the `myPV::write()` function of the PV class.

Setting Up a Server Tool

A server tool must derive a class from the `caServer` class and provide implementations for the `pvExistTest()` and `createPV()` functions, though it can provide implementations for the other virtual functions that are part of the `caServer` class as well as functions of its own. Any extra members needed for a particular implementation can also be provided as part of the derived class. Our derivation of `caServer` is called `myServer` and contains a constructor, implementations for the `createPV()` and `pvExistTest()` virtual functions, a function called `findPV()`, and another member function called `read()`. Two private members are also part of the class, `pvList` and `funcTable`.

```
// The server class. Provides the pvExistTest() and createPV()
// functions as well as the function table.
class myServer : public caServer{
 public:
  myServer(unsigned pvCountEstimate);
  pvExistReturn pvExistTest (const casCtx &ctx, const char
*pPVName);
  static const pvAttr *findPV(const char* Name);
  pvCreateReturn createPV(const casCtx &ctx, const char *pPVName);
  static gddAppFuncTableStatus read(myPV &pv, gdd &value)
  {
   return myServer::funcTable.read(pv, value);
  }
 private:
  static const pvAttr pvList[];
  static gddAppFuncTable<myPV> funcTable;
};
```

pvExistTest()

`pvExistTest()` is a virtual function defined in the `caServer` base class. It's called by the server library when a client is searching for a PV. When the client broadcasts the PV's name across the network, the server library receives the request and calls `pvExistTest()`. The server tool must provide an implementation of `pvExistTest()`, which should inform the server library which will inform the client that the PV exists or doesn't exist in the server tool. It's the responsibility of the server tool to determine if the PV exists or not.

Exactly how a server tool determines if a PV exists doesn't matter. Although if a server tool keeps track of many PVs, an efficient algorithm such as a hashing algorithm is recommended. `pvExistTest()` accepts two arguments as we can see from its prototype:

```
 virtual pvExistReturn pvExistTest (const casCtx &ctx, const char
*pPVAliasName);
```

The first argument is a `casCtx` or "client context" object. This type of object appears in many of the calls in the API. It's a handle used by the internals of the server library, but the server tool should never have to deal with it directly, except to make sure it's passed to functions or constructors that require it. Here, it can be ignored. The second argument, `pPVName`, is a string that holds the name of the PV that the client is interested in. This is the name that the server tool's search algorithm must use to find the PV.

Strictly speaking, the `pvExistTest()` should return an object of the `pvExistReturn` class. `pvExistReturn` is a class whose purpose is solely for communication between the `pvExistTest()` function and the server interface. As mentioned above, it's a container class for an enumerated type, `pverExistReturnEnum`, whose three enumerators are `pverExistsHere`, `pverDoesNotExistHere`, and `pverAsyncCompletion`. A `pvExistReturn` object whose enumerated type has the value `pverExistsHere` indicates to the server library that the PV was found, while the enumerator `pverDoesNotExistHere` indicates that it wasn't. The enumerator `pverAsyncCompletion` tells the server library that the server tool wants to complete the task later. (See *Chapter 3: The casChannel Class and the Asynchronous IO Classes* for more information on asynchronous completion.)

As also mentioned before, the server tool doesn't have to explicitly initialize a `pvExistReturn`. Instead, `pvExistTest()` can simply return the appropriate enumerated value to the server library, and a `pvExistReturn` object initialized with the specified value will be returned to the server library, indicating the existence or nonexistence of the PV. Thus, since our server tool doesn't support asynchronous completion, our version of `pvExistTest()` merely returns either `pverExistsHere`, or `pverDoesNotExistHere`.

The basic algorithm is

```
    if PV Exists
```

```
        return pverExistsHere.
    else
        return pverDoesNotExistHere.
```

Let's take a look at our implementation of `pvExistTest()`. In the `myServer` class there is a private member called `pvList[]` that is an array of `pvAttr` objects. Remember that the `pvAttr` class is simply a way to keep track of PVs.

```
pvExistReturn myServer::pvExistTest(const casCtx &ctx,   //CA
Context
                                                     const char
*pPVName, // PV name
{
  const pvAttr *pPVAttr;
  // If the PV exists, write its name to the canonical PV name
object.
  pPVAttr = myServer::findPV(pPVName);
  if (pPVAttr)
    return pverExistsHere;
  else
   return pverDoesNotExistHere;
}
```

Basically, our version of `pvExistTest()` creates a pointer to a `pvAttr` object. It then calls `findPV()`, a function that simply searches through the elements of the `pvList` array,

```
const pvAttr *myServer::findPV(const char *pName)
{
  const pvAttr *pPVAttr;
  int i = 0;
  short nelem = NELEMENTS(myServer::pvList);

  for(pPVAttr = myServer::pvList; i < nelem; i++, pPVAttr++){
    if (strcmp(pName, pPVAttr->getName().string()) == 0)
      return pPVAttr;
  }
  return NULL;
}
```

comparing the name of each `pvAttr` object to the name passed to it. If the PV exists in the `pvList[]` array, `findPV()` returns a pointer to the `pvAttr` object; otherwise it returns NULL.

Note the **if** statement in the **for** loop. Because the `name` member of `pvAttr` is, in fact, an `aitString` object instead of a character array like other strings, it must be accessed differently if it is to be used as a character array, the type of argument expected by `strcmp()`. The `string()` member function of the `aitString` class accesses the character string contained within the class. This function should be used when interfacing `aitString` objects to functions that expect strings to be character arrays instead of class objects.

If `findPV()` returns a pointer to a `pvAttr` object, `pvExistTest()` returns a `pverExistsHere`. Otherwise, if `findPV()` returns NULL, our version of `pvExistTest()` returns a `pverDoesNotExistHere`.

createPV()

After the server library confirms that the PV exists within the server toolÄthat is, if `pvExistTest()` returns `pverExistsHere`Äthe server library calls `createPV()`. Thus, `createPV()` is called by the server library once and only once for each PV that exists on the server tool if at least one client requests a connection to the PV.

The basic task of `createPV()` is to return either a `casPV` object, a pointer to a `casPV` object, or a status code indicating why a `casPV` object wasn't returned (since the `casPV` class is an abstract class, the expected object should be a pointer to a derived class). Actually, the return value of `createPV()` is a `pvCreateReturn` object which is just a container for a `casPV` object and a status code. However, by simply returning a `casPV` object or a status code, a `pvCreateReturn` object is returned to the server library initialized with the appropriate values. Here's a summary of the possible return values:

- `*casPV`Äsuccess (pass by pointer)
- `casPV`Äsuccess (pass by reference)
- `S_casApp_pvNotFound`Äno PV by that name here
- `S_casApp_noMemory`Äno resource to create pv
- `S_casApp_asyncCompletion`Ädeferred completion

If a pointer to a `casPV` object or a `casPV` object is returned, the status code in the `pvCreateReturn` object will be `S_casApp_success`. However, if the pointer is NULL, the status code will be `S_casApp_pvNotFound`. The last three values that can be returned are error codes. When these are returned, the `pvCreateReturn` will contain the status code, and its pointer to the `casPV` object will be NULL.

Theoretically, `createPV()` should create an object of a class derived from `casPV` and return it to the server library. However, it's possible to simply precreate all the necessary PV objects before `createPV()` is calledÄwhen the server tool is initialized for instanceÄand then return a pointer to one of these objects in `createPV()`. One of the arguments to `casPV` is the name of the PV. The server tool can use this name to determine which PV object should be returned to the server library.

Here's the prototype:

```
virtual pvCreateReturn createPV (const casCtx &ctx, const char
*pPVAliasName);
```

One again, the `casCtx` object can be ignored, but the second argument, the character string representing the PV's name, should be used to confirm that a PV by this name exists. Our implementation of `createPV()` calls `findPV()`, which returns a pointer to a `pvAttr` object if the PV exists or else returns NULL. If NULL is returned by `findPV()`, then `createPV()` returns the status code `S_casApp_pvNotFound`. Otherwise, it calls the new operator to create a `myPV` object on the free store. If a `myPV` object is successfully allocated, the pointer to this object is returned. Otherwise, a memory problem is assumed to have occurred and the status `S_casApp_noMemory` is returned to the server library.

```
pvCreateReturn myServer::createPV(const casCtx &ctx, const char
*pPVName)
{
  const pvAttr *pAttr;
  myPV *pPV;
  // If PV doesn't exist, return NULL. Otherwise, create PV
object.
  pAttr = myServer::findPV(pPVName);
  if (!pAttr)
    return S_casApp_pvNotFound;
  pPV = new myPV(*this, *pAttr);
  if(pPV)
   return S_casApp_noMemory;
  else
   return pPV
}
```

Whether PV objects are created in `createPV()` or before `createPV()` is called, the PV object must exist after `createPV()` returns. Thus, creating the PV object inside `createPV()` as an automatic variable will not work, since the server library must have a way to access the PV object in order to carry out any client-requested operations.

Other myServer Members

The other members of the `myServer` class are the constructor `myServer()`, the function `read()`, and the private member `funcTable`. All these will be discussed when we discuss IO operations below.

The casPV Class

As part of our server tool, we have derived a new class from the `casPV` class, `myPV`. Since the `casPV` class is an abstract class, your server tool will want to derive a class. In addition, many of the default versions of its virtual functions are really not useful. For instance, the `read()` function is a virtual function whose default simply returns the status code `S_casApp_noSupport`. Such default implementations are provided so that a server tool may ignore those functions it doesn't wish to use.

Each PV must have a PV object created for it. This can be done in the `caServer::createPV()` function which is called once for each PV in the server tool (provided that at least one client has requested a connection with the PV). The `casPV` class contains the functions that are responsible for IO, that is, reading and writing to a PV, as well as implementing monitors on a PV. Thus, the core functions are the `read()` and `write()` functions provided as virtual functions in the `casPV` class and which you will want to redefine if you want your server tool to perform IO. To be able to implement an IO function you should have some knowledge of the `gdd` library and its associated classesÄ`gdd`, `gddScalar`, `gddAtomic`, and `gddContainer`.

casPV()

The constructor for the `casPV` class has only one argument: an object of the `caServer` class which is supposed to be the current server object, the server associated with the PV. No matter what other tasks the constructor for a class derived from casPV class does, it must pass the current server's object to the `casPV` constructor. Our server tool creates a `myPV` object in the `createPV()` function, passing to it the current object (`*this`) and a `pvAttr` object.

```
      pPV = new myPV(*this, *pAttr);
```

The `myPV` constructor passes the myServer object to the `casPV` constructor. Here is a representation of the call hierarchy that creates the `casPV` object:

```
                  server library
                                              |
                                              |
                       \/
pvCreateReturn createPV(const casCtx &ctx, const char *pPVName);
                                              |
                                              |
                       \/
  myPV(const caServer &cas, const pvAttr &attributes);
                                              |
                                              |
                       \/
                 casPV (caServer &cas);
```

What the `casPV` constructor actually does should not be the programmer's concern, only that the right arguments are passed to it. The constructor of the derived class, `myPV`, other than initializing the `casPV` class, gives the value of the PV an initial value. Let's take a glance at `myPV()`.

```
myPV::myPV(const caServer &cas, const pvAttr &attributes) :
  attr(attributes),
  casPV(cas),
  interest(aitFalse)
{
  double value;                    // The initial value of the PV.
  gdd *pValue =  attr.getVal();    // Get pointer to gdd object.
  if(!pValue)
```

```
    return;

  // rand() is used to generate a random number from 0 to
RAND_MAX.
  // This number is made to fit in the 0 - 100 range.
  value = (double)rand();
  while (value < 100.0)
    value/=100.0;

  // Use the gdd::putConvert() function to put the value in the
gdd
  // object. Then use the gdd::setStat() and gdd::setSevr() to set
  // the appropriate status and severity for the PV.
  pValue->putConvert(value);
  if (value >= 95){
    pValue->setStat(epicsAlarmHigh);
    pValue->setSevr(epicsSevMinor);
  }
  else if (value <=5 ){
    pValue->setStat(epicsAlarmLow);
    pValue->setSevr(epicsSevMinor);
  }
  else {
    pValue->setStat(epicsAlarmNone);
    pValue->setSevr(epicsSevNone);
  }
}
```

Firstly, note that the `casPV` constructor is properly initialized by passing to it the `myServer` object .

```
  casPV(cas), // initialize base class
```

Also initialized by the constructor are the `attr` and `interest` members of the `myPV` class.

```
  attr(attributes),
  interest(aitFalse)
```

The `attr` member is an object of the `pvAttr` class. The objects of the `pvAttr` class will keep track of the PV's value and its attribute. The `interest` member is initialized to `aitFalse`. We will discuss the significance of the `interest` member when we discuss monitors.

The rest of our constructor gives an initial value to the PV's value: a random number is generated using `rand()`. If this number is greater than 100, it is divided by 100 until it is within the 0-100 range.

```
value = (double)rand();
while (value > 100.0)
    value/=100.0;
```

The value is written to the gdd object using the `putConvert()` function, which is a member function of the `gdd` class:

```
pValue->putConvert(value);
```

The `putConvert()` function writes the value into the `gdd` object; it also converts the value to the object's primitive type. It may not always be obvious what the primitive type is. In our server tool, we know it's aitEnumFloat64. Depending on the architecture of the machine the server tool is running on, this may or may not correspond to type double, the type of the value being written into the `gdd` object.. Therefore, we may or may not have to convert the value to the primitive type of a gdd object. If the server tool is going to be compiles on different architectures, the best way to write a value to a `gdd` object is to use the `gdd::putConvert()` function which will convert the type of its argument to the object's primitive type, if the types are different. If we used `gdd::put()` function instead of `gdd::putConvert()`, the value would be written into the `gdd` object as is, and the primitive type of the object would be changed to reflect the type of the new value.

The rest of the constructor sets the status and severity of the PV. All `gdd` objects have a member which can keep track of both the status and severity of a PV. This member is of type `aitStatus`, defined in `aitTypes.h`. The severity and status can be set by using the `setStat()` and `setSevr()` functions. In our constructor, if the value is over 95 we set the status to `epicsAlarmHigh` and the severity to `epicsAlarmMinor`. If the value is under 5, we set the status to `epicsAlarmLow` and the severity to `epicsAlarmMinor`. Otherwise, we set the status to `epicsAlarmNone` and the severity to `epicsSevNone`. These status and severity levels are taken from the header file alarm.h and are EPICS-specific. You can provide your own severity codes if you wish, but must remember that the `aitStatus` type is an unsigned integer, so severity codes such as -1 or 2.5 are incompatible with the `gdd` library.

Reading PV Values and Their Attributes

The `casPV::read()` function is a virtual function that the server library calls when a client requests to read a value from a PV already attached to the server tool, that is, a PV for which a class derived from `casPV` has already been created. It has two arguments. The first is a `casCtx` object which can be ignored. The second argument is a `gdd` object. It is the main task of the `read()` function to write the current value of the PV into this `gdd` object. Note that this `gdd` object is passed by reference to `read()`. Thus, after `read()` returns, the server library can then retrieve the current value of the PV from the `gdd` object and forward this value to the client.

If the operation is successful, `read()` should return the success status code, `S_casApp_success`. Otherwise, it should return the appropriate error code. The status code `S_casApp_asyncCompletion` can also be returned if the server tool supports asynchronous completion for that PV. See *Chapter 3: The casChannel Class and the Asynchronous IO Classes* for more information on asynchronous completion.

Here is the prototype for `read()`:

```
virtual caStatus read (const casCtx &ctx, gdd &prototype);
```

What complicates satisfying read requests is that a server tool may have to satisfy requests for various types. The second argument, `prototype`, should specify an application type. An EPICS client makes requests using database request types such as `DBR_INT` or `DBR_STS_INT`. When the server library receives a client request, it maps it to a `gdd` object or objects and assigns to each object an application type. For instance, if an existing EPICS client makes a request to read the value of a PV using the request type `DBR_STS_DOUBLE`, the server library will receive the request and map it to an application type or types. Since `DBR_STS_DOUBLE` is a request for three valuesÄthe PV's value, its status, and its severityÄthe server library will map the request to a `gddContainer` object and flag the object as the `dbr_sts_double` application type. The server library will then assign three gdd objects to the container object: one will be of the "value" application type for the PV's value, another will be of the "status" application type for the PV's value, and the last will be of the "severity" application type for the PV's severity. In this way, all requests are mapped to gdd objects and assigned application types. Compound DBR types like `DBR_STS_DOUBLE` are mapped to `gddContainer` objects, while other DBR types like `DBR_DOUBLE` are mapped to either `gddScalar` or `gddAtomic` objects.

The server library then forwards the `gdd` object to the `read()` function via `prototype`, which is passed by reference to `read()` as the second argument. By checking the application type of the object, the server tool can tell which application type or types it's dealing with, and write the appropriate value or values into the `gdd` object.

Thus, quite a burden is placed on `read()` to be able to deal with requests for multiple application types, and yet at the same time deal with simple requests for a single application type. There are several ways to do this. One way is simply check to see if `prototype` is a container, and if so step through the container writing the appropriate values to the appropriate `gdd` objects in the container. Most of the functions to do this are already provided. However, another, simpler way is to create a *function table*, which is the approach our server takes. To do this, you use the `gddAppFuncTable<>` template in the header file gddAppFuncTable.h.

Our function table is a private, static member of the `myServer` class. Here is its declaration:

```
    gddAppFuncTable<myPV> funcTable;
```

The template is initialized for `myPV` objects. A function table is basically an array of function pointers. A function is installed for each application type. When the `gddAppFuncTable::read()` is called, it is passed a pointer to the `myPV` class which called it and the `gdd` object. The `gddAppFuncTable::read()` function will step through the `gdd` prototype if it's a container and call the appropriate functions corresponding to the application types of the `gdd` objects.

For instance, suppose we wanted our server to read `DBR_GR` requests. For all such `DBR_GR` requests, the gdd prototype passed to `casPV::read()` would be a container. The application type of the container would be `gddAppType_dbr_gr_short` or `gddAppType_dbr_gr_float` or whatever. The container, in turn, would consist of several `gdd` objects, each of which would have its own application type such as `gddAppType_status` or `gddAppType_seconds` (see gddApps.h). If this object is passed to `gddAppFuncTable::read()`, `gddAppFuncTable::read()` calls the appropriate function from the function table for each application type in the object.

To install a function in the function table, use the `gdd::installReadFunc()` function, and as its first argument pass an application type and a function that corresponds to that application type, i.ei., the address of the function that retrieves the value for that application type. The application type can be either a constant such as those from gddApps.h or a string describing a valid application type, such as the default application types in gddAppDefs.h. For example, to install a function to read the PV's status, we would pass the string ❽tatus❾as the first argument to `installReadFunc()` and the address of the function as the second argument.

Let's see how our server tool does this. Since we want our server tool to satisfy DM requests and we know that DM makes an initial `DBR_CTRL` request for each PV it wishes to connect to, our server tool must create a function for each application type in the `DBR_CTRL` structure and install that function in the function table. All these functions are members of the `myPV` class. For example, one of the members of `DBR_CTRL` is `upper_warning_limit`, a value that in the EPICS database comes from the `HIGH` field of a record. The `DBR_CTRL` structure is mapped to a `gdd` container object that is of the appropriate application type, `gddAppType_dbr_ctrl_float`. The members of the `DBR_CTRL` structure are, in turn, each mapped to a `gdd` object in the container, each identified by its own application type. The application type for the `upper_warning_limit` member of the DBR_CTRL structure is `gddAppType_AlarmHighWarning`. Our function `readHighAlarm()`, will write a value into the `gdd` object that is identified by `gddAppType_AlarmHighWarning`. Here is the function itself:

```
gddAppFuncTableStatus myPV::readHighAlarm(gdd &value)
{
  value.putConvert(attr.getHighAlarm());
  return S_casApp_success;
}
```

Each function that is installed in a function table must return the status code of type `gddAppFuncTableStatus`, which is basically the same type as `caStatus`. The main task of all such ❽ead❾functions is to write a value into the `gdd` object passed to them. Thus, our function uses the `putConvert()` function, discussed above, to write the value of the high warning limit of the PV into the gdd object. The other read functions such as `readStatus()` appear in Appendix A.

The functions must be installed in the function table. Our server chooses to install the functions in the constructor to the `myServer` class:

```
// Constructor for myServer. After passing arguments to caServer
// constructor all of the read functions are installed in the
function
// table.
 myServer::myServer(unsigned pvCountEstimate) :
 caServer(pvCountEstimate)
{
   funcTable.installReadFunc("status", myPV::readStatus);
  funcTable.installReadFunc("severity", myPV::readSeverity);
  funcTable.installReadFunc("precision", myPV::readPrecision);
  funcTable.installReadFunc("alarmHigh", myPV::readHighAlarm);
  funcTable.installReadFunc("alarmHighWarning",
myPV::readHighWarn);
  funcTable.installReadFunc("alarmLowWarning", myPV::readLowWarn);
  funcTable.installReadFunc("alarmLow", myPV::readLowAlarm);
  funcTable.installReadFunc("value", myPV::readValue);
  funcTable.installReadFunc("graphicHigh", myPV::readHopr);
  funcTable.installReadFunc("graphicLow", myPV::readLopr);
  funcTable.installReadFunc("controlHigh", myPV::readHighCtrl);
  funcTable.installReadFunc("controlLow", myPV::readLowCtrl);
  funcTable.installReadFunc("units", myPV::readUnits);
}
```

The first argument to the calls to `installReadFunc()` is the character string equivalent of an application type, though constants can be passed as the first argument. These names can be found in the header file gddAppDefs.h and are default application types for EPICS-type data.

Thus, when a client makes a `DBR_CTRL` read request of our server tool, the server library will call `myPV::read()`, a redefinition of the `casPV::read()` virtual function:

```
caStatus myPV::read(const casCtx &ctx, gdd &prototype)
{
  // Calls myServer::read() which calls the appropriate function
from
  // the application table.
  return myServer::read(*this, prototype);
}
```

All `myPV::read()` does is to call `myServer::read()` whose definition is

```
  static gddAppFuncTableStatus read(myPV &pv, gdd &value)
  {
      return myServer::funcTable.read(pv, value);
  }
```

which calls `gddAppFuncTable`::`read()`. This function accepts two arguments: the first is an object of the type for which the template was instantiated, a `myPV` object in this case. Its second argument is a `gdd` object. It will then call the functions for the application type(s) in the gdd object. Thus, for `DBR_CTRL` requests all the functions installed in the function table will be called, each writing the requested value into the gdd object.

If the request, on the other hand, were only for the PV's value, then the same thing would occur, but only the function for the ❽value❾application type, myPV::readValue() would be called. Let's take a look at the readValue() function:

```
gddAppFuncTableStatus myPV::readValue(gdd &value)
{
  // If pvAttr::pValue exists, then use the gdd::get() function to
  // assign the current value of pValue to currentVal; then use
the
  // gdd::putConvert() to write the value into value.
  gdd *pValue = attr.getVal();
  double currentVal;
  if(!pValue)
    return S_casApp_undefined;
  else {
    pValue->getConvert(currentVal);
    value.putConvert(currentVal);
    return S_casApp_success;
  }
}
```

When readValue() is called from the function table, it is passed a gdd object. The server library expects the current value of the PV to be written into this object. Our function makes a pointer to the gdd object member of the attr, which is a pvAttr class object. After making sure that the pointer is not NULL, the value of the gdd object now pointed to by pValue is retrieved using the getConvert() member function. getConvert() is a member function of the gdd class. It accepts by reference any valid **lvalue** (variable) as an argument, and assigns to it the current value of the PV.

readValue() then writes the current value of the PV into the gdd object value using putConvert() and passing to it the same lvalue just passed to getConvert(). getConvert() will convert the data of the gdd object to the data type of the lvalue passed to it, converting strings, for instance, to their numerical representation. putConvert(), as mentioned earlier, converts the data of the value passed to it to the primitive data type of the gdd object's data.

Note that our function returns an error code, S_casApp_undefined, if an error occurred, and the success code, S_casApp_success if the operation was successful.

Thus, the call hierarchy would look like this when a client requests a PV's value:

```
                    Server Library
                                                |
                                                |
                       \/
   caStatus myPV::read(const casCtx &ctx, gdd &prototype)
                                                |
                                                |
                       \/
static gddAppFuncTableStatus myServer::read(myPV &pv, gdd &value)
                                                   |
                                                   |
                       \/
gddAppFuncTableStatus gddAppFuncTable::read(PV &pv, gdd &value)
                                                   |
                                                   |
                       \/
gddAppFuncTableStatus myPV::readValue(gdd &value)
```

Although this my seem like great lengths to go to read a value, when one considers the complexity involved in dealing with different data types, converting between those data types, and dealing with compound data types, that is, requests for multiple values, much of the complexity has been reduced by the `gdd` library and the server library. For example, our server, even though it consists of relatively little code, can satisfy read requests by most existing EPICS clients.

Writing Values to PVs

Currently, no existing EPICS clients can make Channel Access ❽put❾requests using compound types like `DBR_STS` or `DBR_GR`. That is, by using a `DBR_STS` request type, for instance, a client cannot change the PV's value, status, and severity. Although possible, implementing the ability to make such requests is of limited value for the simple fact that clients don't need to change a PV's attributes that often, and when they do, it is possible, at least in an EPICS database, to simply connect to the field that holds the attribute. The attribute field then simply becomes its own PV and can be changed as any other PV. In our server, this won't work for there is no way to attach to an attribute as its own PV, but in an EPICS database this is possible.

Thus, when writing to a PV, a server tool should, as yet, not be concerned with writing a set of values.

Any fully functional server tool must have the ability to read and write arrays or sub-arrays. Writing and reading arrays with the server tool brings up several complications which will be covered in a subsequent chapter.

Thus, the task of our `write()` function is very simple: to write a single, scalar value to a PV.

Like `casPV::read()`, `casPV::write()` is a virtual function which a server tool can redefine in a derived class and will probably want to because the default version simply returns the error code `S_casApp_noSupport`. The server library calls `write()` when a client wants to write a value to a PV. The `write()` function accepts the same arguments as the `read()` function, a `casCtx` object and a `gdd` object. Here is the prototype:

```
   virtual caStatus casPV::write (const casCtx &ctx, gdd &value);
```

The basic task of the write function is to copy or reference the data of the `gdd` object, `value`, to the PV.

If `value` contains a large array, to copy it would represent a significant amount of memory lost. However, one could merely reference it, that is, set a gdd-type pointer to point to the data, thus not using up that amount of memory. However, this approach is only possible if your server tool is storing the value using a gdd or derived classÄ`gdd`, `gddScalar`, `gddAtomic`, or `gddContainer`. Otherwise, you will have to copy the data. For scalar values, simply copying the data is efficient. Since our function only deals with writing scalar values, this is what we will do.

The other task of the `write()` function is to return the appropriate status code indicating success or error.

Here is our `write()` function as redefined in myPV:

```
caStatus myPV::write(const casCtx &ctx, gdd &value)
{
  struct timespec t;
  osiTime current(osiTime::getCurrent());
  gdd *pValue;
  caServer *pServer = this->getCAS();
  double newVal;

  // Doesn't support writing to arrays or container objects
  // (gddAtomic or gddContainer).
  if(!(value.isScalar()) || !pServer)
    return S_casApp_noSupport;

  pValue = attr.getVal();
  // If pValue exists, unreference it, set the pointer to the new
gdd
  // object, and reference it.
  if(pValue)
    pValue->unreference();
  pValue = &value;
  pValue->reference();

  // Set the timespec structure to the current time stamp the gdd.
  current.get(t.tv_sec, t.tv_nsec);
  pValue->setTimeStamp(&t);

  // Get the new value and set the severity and status according
  // to its value.
  value.get(newVal);
  if (newVal > 100){
    value.setStat(epicsAlarmHiHi);
    value.setSevr(epicsSevMajor);
  }
  else if (newVal >= 95){
    value.setStat(epicsAlarmHigh);
    value.setSevr(epicsSevMinor);
  }
  else if (newVal <=5 ){
    value.setStat(epicsAlarmLow);
    value.setSevr(epicsSevMinor);
  }
  else if (newVal < 0){
    value.setStat(epicsAlarmLoLo);
    value.setSevr(epicsSevMajor);
  }
  if(interest == aitTrue){
    casEventMask select(pServer->valueEventMask |
                                                pServer-
>alarmEventMask);
    postEvent(select, *pValue);
  }
  return S_casApp_success;
}
```

Let's look at the declarations first. `t` is a `timespec` structure that will be used to timestamp the value being written. In addition to having a member to hold the value's status, all `gdd` classes have a way to keep track of a timestamp.

The next declaration is for an `osiTime` object, `current`, which we will use to assign the current time to the `timespec` structure. The `osiTime` or *operating-system-independent timer* class, is an easy way for a server tool to keep track of time on a system. Its declaration is in the osiTimer.h header file.

`pValue` will of course reference the PV's value, the `gdd` object contained by the pvAttr class. `pServer` will point to the server to which this PV is attached. We do this by calling `getCAS()` which is a member function of the `casPV` class that returns a pointer to the server object associated with the current PV. We will see the reason for doing this shortly. And we'll assign to `newVal` the actual numerical value of the new value to be written so that we may check it and determine the severity of the PV.

Our first task is to make sure that the value passed to `write()` is a scalar value, for our server tool only supports writing a single scalar value as opposed to an array of values or a container of `gdd` object. We can do this using the `isScalar()` function, part of the `gdd` class. At the same time we'll make sure that there actually is a server attached to this PV.

```
 if(!(value.isScaler()) || !pServer)
    return S_casApp_noSupport;
```

Next, we set `pValue` to point to the PV's value,

```
  pValue = attr.getVal();
```

If `getVal()` returns a pointer to a gdd object and not NULL, the referenced object is ❽ unreferenced❾, the pointer is set to point to the new object, and the new object is then ❽ referenced.❾

```
  if(pValue)
    pValue->unreference();
  pValue = &value;
  pValue->reference();
```

It is possible for a `gdd` object to be referenced by multiple applications like multiple servers tools at the same time. For instance, in the above example, `value` may also be referenced by another server tool at the same time that it is referenced by our server tool. However, each server tool must indicate that it is referencing the `gdd` object using the `gdd::reference()` function, which increments the reference count by one. When a server tool wishes to indicate that it has no more interest in the gdd object, it should call `gdd::unreference()`, which decrements the reference count by one. When the reference count reaches zero, the `gdd` object is deleted. It's important that a server tool not attempt to destroy or delete a `gdd` object explicitly. Instead, by using `reference()` and `unreference()` a server tool can use the gdd library's own internal mechanism for destroying classes.

Another way to write to a PV's value would be simply to copy the information from `value` to `pValue`, or to simply get the value from `value` and write it into `pValue` as in the following bit of code:

```
  pValue = attr.getVal();
  value.getConvert(newVal);
  pValue->putConvert(newVal);
```

In this case since we are not referencing the new value, we don't have to call `reference()` or `unreference()` for either `pValue` or `value`. Either way of implementing `write()` will work, but each has its advantages and drawbacks.

The rest of our code timestamps the new value,

```
 current.get(t.tv_sec, t.tv_nsec);
 pValue->setTimeStamp(&t);
```

and then sets the status and severity of the value depending on the value.

```
  value.get(newVal);
  if (newVal > 100){
    value.setStat(epicsAlarmHiHi);
    value.setSevr(epicsSevMajor);
  }
  else if (newVal >= 95){
    value.setStat(epicsAlarmHigh);
    value.setSevr(epicsSevMinor);
  }
  else if (newVal < 0){
    value.setStat(epicsAlarmLoLo);
    value.setSevr(epicsSevMajor);
  }
  else if (newVal <= 5 ){
    value.setStat(epicsAlarmLow);
    value.setSevr(epicsSevMinor);
  }
  else{
    value.setStat(epicsAlarmNone);
    value.setSevr(epicsSevNone);
  }
```

All the status codes and alarm codes in the above code are EPICS-specific and appear in the alarm.h header file.

Monitoring PVs

The following bit of code from `write()` then posts a monitor for the value:

```
if (interest == aitTrue){
    casEventMask select(pServer->valueEventMask |
                        pServer->alarmEventMask);
    postEvent(select, *pValue);
 }
```

The server library calls `casPV::interestRegister()` when a client wishes to subscribe for monitor events from the server tool for a particular PV. The server library calls `casPV::interestDelete()` when a client wishes to remove its monitor for that PV. Usually, this means that `interestRegister()` will be called when the first clients has requested to monitor the PV and that `interestDelete()` will be called when the last client has removed its monitor.

`interestRegister()` and `interestDelete()` are virtual functions of the `casPV` class whose default implementations are ❸mpty❾functions that perform no real task. Thus, if a server tool is going to support monitors for its PVs, it must provide an implementation of both functions in a derived class. Typically, the functions will provide a way for a server tool to keep track of whether or not there are monitors on this PV. Our version of `interestRegister()` merely sets the `interest` member of the `myPV` class to `aitTrue`. `interest` is a variable of type `aitBool`, an enumerated type whose enumerators are `aitTrue` and `aitFalse`. When interest is `aitTrue`, it means that a client or clients have established monitor(s) on the PV. Our version of `interestDelete()` sets `interest` to `aitFalse`. When interest is `aitFalse`, it means that no more clients have monitors on this PV. Here are the implementations for `interestRegister()` and `interestDelete()` as can be seen in the class declaration for `myPV`. Note that `interestRegister()` returns a status code, while `interestDelete()` does not.

```
 caStatus interestRegister()
 {
   interest = aitTrue;
   return S_casApp_success;
 }

 void interestDelete() { interest = aitFalse; }
```

The server tool is responsible for posting all monitors on a PV. An ideal time to do this is when the PV's value is changed, as in a `write()` operation. Thus, `write()` checks to see if monitors have been established for this PV,

```
if (interest == aitTrue){
```

and if so, posts a monitor event for the PV using `casPV::postEvent()`. `postEvent()` is a member of the `casPV` class and can be used by a server tool to inform the server library that the PV's value has changed; the library then notifies the client that a monitor event occurred. `postEvent()` accepts an object of the `casEventMask` class as its first argument and a `gdd` object containing the PV's new value as its second argument.

```
 void postEvent (const casEventMask &select, gdd &event);
```

`casEventMask` objects simply provide a way to represent event masks by the server library and to combine them by ORing or to unmask them by ANDing them. Thus, we create a `casEventMask` object called `select` and pass as an argument two `casEventMask` objects ORed together. These two `casEventMask` arguments are public members of the `caServer` class and represent two of the three types of masks currently available in the Portable Server API. Here they are as they appear in `caServer`:

```
   const casEventMask valueEventMask; // DBE_VALUE
   const casEventMask logEventMask;  // DBE_LOG
   const casEventMask alarmEventMask; // DBE_ALARM
```

The first mask, `valueEventMask`, is a mask for value-change events; the second, `logEventMask`, for archival-change events; and the third, `alarmEventMask`, for alarm-change events. However, we choose to merely implement the first two for this PV. Thus, the following code, posts value-change and archival monitors for our PV:

```
if (interest == aitTrue){
    casEventMask select(pServer->valueEventMask |
                        pServer->alarmEventMask);
    postEvent(select, *pValue);
 }
```

Lastly, `write()` returns the success error code:

```
return S_casApp_success;
```

There are, of course, numerous other ways to implement `casPV::write()`, but the above is a simple and obvious one.

beginTransaction () and endTransaction()

There are other virtual functions that form part of the `casPV` base class. The two that haven't already been discussed are `beginTransaction()` and `endTransaction()`, both of which are virtual functions whose default implementations perform no real task. Neither accepts any arguments.

`beginTransaction()` must return a status code, while `endTransaction()` returns nothing. Here are their prototypes:

```
virtual caStatus beginTransaction ();
virtual void endTransaction ();
```

The server library calls `beginTransaction()` before it calls either `casPV::read()` or `casPV::write()` (or the redefinitions of them in a derived class). The server library calls `endTransaction()` after it calls either `casPV::read()` or `casPV::write()`. Neither function has to perform a specific task, but can do whatever the server tool needs them to do. For instance, our implementation of `beginTransaction()` merely increments the counter `currentOps`:

```
caStatus myPV::beginTransaction()
{
  // Trivial definition that informs the user of the number of 
current
  // IO operations in progress for the server tool. currentOps is 
a
  // static member.
  currentOps++;
  cerr<<"Number of current operations = "<<currentOps<<"\n";
  return S_casApp_success;
}
```

`currentOps` is a static member of `myPV`. Since it is a static member, its value will be common to all `myPV` objects and hence to all PVs. Hence, it will keep track of the total number of operations in progress on all PVs in this server tool. `S_casApp_success` is then returned. `endTransaction()` decrements the counter `currentOps` after each `read()` or `write()` function is called.

That is our simple server tool. The code in its entirety can be found in Appendix A. The next chapter will discuss how to deal with the `casAsyncReadIO`, `casAsyncWriteIO`, `casAsyncPVExistIO`, casAsyncCreatePVIO, and the `casChannel` class.

Chapter 3: The casChannel Class and the Asynchronous IO Classes

The `casChannel` class and the asynchronous IO classes: `casAsyncReadIO`, `casAsyncPVExistIO`, `casAsyncCreatePVIO`, and `casAsyncWriteIO`Äare optional classes: a server tool only needs to use the `casChannel` class if it's going to implement some type of access control, and it only needs to use the asynchronous IO classes if it's going to support some type of asynchronous IO completion. Since it's anticipated that most server tools will probably not want to implement either access rights or asynchronous completion, most server tools will not need these classes. Nevertheless, they are provided in case a server tool wishes to provide full functionality.

This chapter will explain these classes and how to use them. As an example, a simple program is provided which is just a revision of the program in the previous chapter, with the added capability of asynchronous IO and access control. The listing for this program can be found in Appendix B.

1. Asynchronous IO

The `casAsyncReadIO` class provides a server tool with the ability to delay satisfying a read request, completing the request later. The `casAsyncWriteIO` class provides the server tool with the ability to delay satisfying a write request when `write()` is called. The `casAsyncCreatePVIO` provides the server tool with the ability to delay creating a PV when `createPV()` is called. The `casAsyncPVExistIO` provides the server tool with the ability to delay informing the server library whether or not a PV exists in the server tool when `pvExistTest()` is called. There may be several reasons a server tool might want to delay completing such requests:

- For read operations, the requested value may not be immediately available.
- For write operations, the server tool may have to wait until another application or server tool has completed its own write operation.
- For all operations, it's more efficient if a server tool satisfies read or write operations in batches. When passing through software layers and when being sent over the network, requests are more efficient in batches rather than one at a time.

However, these are only the most obvious reasons. The server library is not concerned with why the server tool wishes to use asynchronous IO: it merely provides a way to ❽ post❾the completion of the operation at a later time.

Asynchronous completion is available for the following three functions:

- `caServer::pvExistTest()`
- `caServer::createPV()`
- `casPV::read()`
- `casPV::write()`

For any of these to complete asynchronously, they must create the appropriate asynchronous IO class and return the appropriate status code. When the operation is complete, `postIOCompletion()` must be called. `postIOCompletion()` is a member of all three asynchronous IO classes, though its exact form is different for each class. How the operation is completed is left up to the server tool.

The basic steps for an asynchronous read operation are,

1. In a redefinition of `casPV::read()`, create a `casAsyncReadIO` or derived-class object and return `S_casApp_asyncCompletion`.
2. When operation is ready to complete, write current value into the `gdd` object passed to `read()`.
3. Call `casAsyncReadIO::postIOCompletion()` with two arguments: a status code and a `gdd` object containing the PV's value.

The basic steps for an asynchronous write operation are,

4. In a redefinition of `casPV::write()`, create a `casAsyncWriteIO` or derived class object and return `S_casApp_asyncCompletion`.
5. When operation is ready to complete, write the value from the `gdd` object passed to `casPV::write()` into the PV.
6. Call `casAsyncWriteIO::postIOCompletion()` with one arguments: a status code, `S_casApp_success` for instance.

In order to complete the second step for asynchronous read and write operations, the server tool must keep track of the `gdd` object passed to `read()` and `write()`. For `read()` operations, the server tool should write the requested values into the `gdd` object, and for `write()` operations, the server tool needs to write the value contained in the `gdd` object to the PV. The best and most efficient way for a server tool to keep the `gdd` object until the operation is ready to be completed is to reference the object with a pointer, and then call `reference()` for the object. If `reference()` is not called, the object will be deleted after either `read()` or `write()` returns.

The basic algorithm for asynchronous completion of `pvExistTest()` is,

7. In a redefinition of `caServer::pvExistTest()`, return the enumerated value `pverAsyncCompletion`.
8. When operation is ready to complete, determine if PV exists.
9. Call `casAsyncPVExistIO::postIOCompletion()` with a `pvExistReturn` object as the argument. This `pvExistReturn` object should be created and initialized with the value `pverExistsHere` or the value `pverDoesNotExistHere`.

Example

We have modified the program presented in the last chapter in order to implement asynchronous read and write operations. Asynchronous completion for the `pvExistTest()` and `createPV()` functions works much the same way.

Basically, we've created two classes:

- `myAsyncReadIO` which is derived from `casAsyncReadIO` and also from a class called `osiTimer`.
- `myAsyncWriteIO` which is derived from `casAsyncWriteIO` and also from `osiTimer`.

`myAsyncReadIO` and `myAsyncWriteIO` have two base classes, `casAsyncReadIO` or `casAsyncWriteIO`, and `osiTimer`. An `osiTImer` object can sleep for a specified number of seconds, after which its member function `expire()` is called. `expire()` is a virtual function which can be redefined to perform any task; we will redefine it to complete the read or write operation when it's called. Thus, for write operations, our algorithm is as follows:

1. When `write()` is called, create a `myAsyncWriteIO` object.
2. In `myAsyncWriteIO` constructor, initialize `osiTimer` object with a sleep time of 10 seconds.
3. When `expire()` is called, unreference current PV value and reference `gdd` object passed to `write()`.
4. Call `postIOCompletion()`, passing to it the status code `S_casApp_success`.

For read operations, the algorithm is similar, but note that a pointer to a `gdd` object is passed to `postIOCompletion()` as the second argument.

5. When `read()` is called, create a `myAsyncIO` object.
6. Initialize `osiTimer` object with a sleep time of 10 seconds.

7. When `expire()` is called, get pointer to `gdd` object passed to `read()` and write current value of PV into gdd object.
8. Call `postIOCompletion()`, passing `S_casApp_success` as first argument and a pointer to the `gdd` object as its second.

Thus, we will have to change the definitions of `myPV::read()` and `myPV::write()`. Here are the new definitions:

```
caStatus myPV::read(const casCtx &ctx, gdd &prototype)
{
  myAsyncReadIO *pIO;
  pIO = new myAsyncReadIO(ctx, prototype, *this);
  if (!pIO)
    return S_casApp_noMemory;
  else
    return S_casApp_asyncCompletion;
}

caStatus myPV::write(const casCtx &ctx, gdd &value)
{
  myAsyncWriteIO *pIO;
  pIO = new myAsyncWriteIO(ctx, value, *this);
  if (!pIO)
    return S_casApp_noMemory;
  else
    return S_casApp_asyncCompletion;
}
```

Basically, the functions create a `myAsyncReadIO` or `myAsyncWriteIO` object using the `new` operator. Each function then makes sure the object was indeed created. If it wasn't, `S_casApp_noMemory`. Otherwise, `S_casApp_asyncCompletion` is returned.

The actual IO is performed by the `expire()` function of the `myAsyncWriteIO` and `myAsyncReadIO` classes. Remember that `expire()` is a virtual function of the `osiTimer` class and it is called after the delay time expires. The `osiTimer` class is initialized with an `osiTime` object. The `osiTime` object is an operating-system independent way to represent time in seconds/nanoseconds. It can be initialized in several waysÄthe easiest way is to pass its constructor a floating-point value representing the number of seconds for the delay. The `osiTimer` object, when initialized, will sleep for the number of seconds represented by the `osiTime` object. When it awakes, it will call the virtual function `expire()`, which is a pure virtual function.

We will redefine `expire()` to perform the actual IO operation. After `expire()` returns, the virtual function `ositTimer::again()` is called, which must return either `aitTrue` or `aitFalse`. If `again()` returns `aitTrue`, the `osiTimer` object will call `delay()`, another virtual function which should return another `osiTime` object representing a time period in seconds. Thus, if `again()` returns `aitTrue`, the `osiTimer` object will sleep again for the time period returned by `delay()`. The default version of `again()` returns `aitFalse`, meaning that the `osiTimer` object is to sleep only once, wake up, call `expire()`, and then destroy itself. Since we want the timer to execute only once, we will not redefine `again()`, only `expire()`.

Here are the class declarations for `myAsyncReadIO` and `myAsyncWriteIO`:

```
class myAsyncReadIO : public casAsyncReadIO, public osiTimer {
 public:
  myAsyncReadIO(const casCtx &ctx, gdd &Value, myPV &pv) :
    casAsyncReadIO(ctx), PV(pv), osiTimer(osiTime(10.0))
    {
      pValue = &Value;
      pValue->reference();
    }
  void expire();
  gdd *pValue;
 private:
  myPV &PV;
};

class myAsyncWriteIO : public casAsyncWriteIO, public osiTimer {
 public:
  myAsyncWriteIO(const casCtx &ctx, gdd &Value, myPV &pv) :
    casAsyncWriteIO(ctx), PV(pv), osiTimer(osiTime(10.0))
    {
      pValue = &Value;
      pValue->reference();
    }
  void expire();
  gdd *pValue;
 private:
  myPV &PV;
};
```

Note the constructors to both classes. Each constructor accepts three arguments: a `casCtx` object, a `gdd` object, and a `myPV` object.

The first two arguments are the arguments that were passed to `read()` or `write()`, a `casCtx` object and a `gdd` object. The first argument must be passed to the constructor for the `casAsyncReadIO` and `casAsyncWriteIO` classes. The second argument is the gdd object passed to `read()`/`write()`. For read operations, the server tool must write the PV's current value into this object. For write operations, the value in this object must be written to the PV. The last argument is the myPV class for which `read()`/`write()` was called. Note that the constructors initialize the `osiTimer` base class by passing to it an `osiTime` object initialized with a value of 10 or ten seconds. Thus, when created, the `myAsyncReadIO` and `myAsyncWriteIO` objects will sleep for ten seconds and, upon waking, will call `expire()`. Note also that the constructors calls `reference()` for the `gdd` object. Thus, after `read()`/`write()` return, the `gdd` object will not be destroyed.

The `myAsyncReadIO::expire()` and `myAsyncReadIO::expire()` functions are simple to follow. They merely perform what would normally be performed in `read()` or `write()`:

```
void myAsyncReadIO::expire()
{
  caStatus status, status1;
  status = myServer::read(PV, *pValue);
  status1 = postIOCompletion(status, *pValue);
  if (status1 != S_casApp_success)
    cerr <<"Error returned by postIOCompletion: "
         <<"myAsyncReadIO::expire()."<<endl;
}

void myAsyncWriteIO::expire()
{
  gdd *pValue1;
  caStatus status;
  pValue1 = PV.getAttr().getVal();

  // Just reference the darn thing!
  if(pValue1)
    pValue1->unreference();
  pValue1 = pValue;
  status = postIOCompletion(S_casApp_success);
  if (status != S_casApp_success)
    cerr <<"Error returned by postIOCompletion: "
         <<"myAsyncWriteIO::expire()."<<endl;
}
```

`myAsyncReadIO::expire()` simply passes the PV and the `gdd` object to the `myServer::read()` function. This function calls the `read()` function of `gddAppFuncTable`, which will call the appropriate functions from the function table needed to read the value from `pValue`. Note that `pValue` is dereferenced (`*pValue`) before it is passed to `myServer::read()`. The status from `myServer::read()` and the pointer to the `gdd` object are then passed to `postIOCompletion()`.

`myAsyncWriteIO::expire()` sets the pointer `pValue1` to the PV's value,

```
pValue1 = PV.getAttr().getVal();
```

If `pValue1` is not NULL, it is unreferenced:

```
pValue1->unreference();
```

Remember that it's possible for more than one application to reference the same `gdd` object. When a server tool first references a `gdd` object by pointer, it should call `reference()`, and when it sets the pointer to point to another object, it should call `unreference()`. `reference()` causes a counter to be incremented, thus keeping track of the number of pointers that are referencing it. `unreference()` decrements the same counter. When the counter reaches zero, the `gdd` object is destroyed. Therefore, a server tool should never directly destroy or delete a `gdd` object that is has referenced.

Our server tool calls `reference()` in the `myAsyncReadIO()` and `myAsyncWriteIO()` constructors: this is so that the `gdd` object will not be destroyed while the operation is waiting to complete. `myAsyncReadIO::expire()` posts the `gdd` object, after which the server library will forward the appropriate values to the client and unreference the object, so the server tool doesn't have to call `unreference()` for read operations. `myAsyncWriteIO::expire()` also references the new value, unreferencing the old one first. Since the gdd object passed to write() represents the new value of the PV, by referencing it we can update the PV's value. This method is an efficient and simple way to deal with arrays and scalar values. However, because the value is simply referenced, no write operations to part of an array are supported.

Next, the status code `S_casApp_success` is passed to `postIOCompletion()`. Note that `casAsyncWriteIO::postIOCompletion()` is not passed a `gdd` object; this is because, for write operations, the client is not interested in a return value, only in the success/failure of the operation.

2. Access Control and the casChannel Class

Access control refers to restricting access to a PV according to the user or to the host or to some combination thereof. For instance, if the user is ❽erry❾and the host is ❽ machine❾, then the user can be granted read access only and not write access. Hence, ❽ terry❾can only read the PV's value and cannot change it. As mentioned in the introduction, a channel refers to the connection between a specific client and a specific PV. Thus, for each client attached to a PV, there is a separate channel. For instance, if two clients establish a connection to the same PV, two separate channels are created, one for each client.

For server tools that don't wish to implement access control, the server library automatically creates a `casChannel` object for each client that establishes a connection to a PV. By default, the `casChannel` class imposes no access control; all clients are given both read and write access. However, by deriving a class from the `casChannel` class and redefining several of its virtual functions, a server tool can implement its own access control. In addition, a server tool that wishes to implement its own access control must redefine `casPV::createChannel()`, a virtual function that by default, creates and returns a `casChannel` object, but which a server tool should can redefine to create a class derived from `casChannel`.

The casChannel Class

When any client establishes a connection to a PV, the server library calls the `casPV::createChannel()` function of the `casPV` class or of a derived class. The primary task of the `createChannel()` function is to create and return a pointer to a `casChannel` object or an object of a derived class. Here is the prototype:

```
virtual casChannel *createChannel ( const casCtx &ctx,
                                                     const char *
const pUserName,
                                                     const char *
const pHostName);
```

It accepts three arguments: a `casCtx` object, and two string pointers, `pUserName` and `pHostName`. `pUserName`, of course, points to a string representing the user's name, and `pHostName` points to a string representing the server tool's host. To implement access control, a server tool will most likely want to keep track of the user and host names. The first argument, `ctx`, must be passed to the constructor for the `casChannel` base class.

Implementing access control with the `casChannel` class is relatively easy. The important functions are:

- readAccess(). This is a virtual function that returns a value of type `aitBool`. Thus, it can return either `aitTrue` or `aitFalse`. By default, it returns `aitTrue`. If `aitTrue` is returned, then that means the current client has permission to read the PV's value and/or its attributes. If `aitFalse` is returned, then that means the current client does not have permission to read the PV's value or any of its attributes.
- `writeAccess()`. The same as `readAccess()`Äa return value of `aitFalse` indicates that the current client doesn't have permission to write to the PV, and a return value of `aitTrue` indicates that it does.

The value returned by `writeAccess()` and `readAccess()` is encouraged to change over a channel's lifetime. That means if `readAccess()` returns, for instance, `aitTrue` at first, it's perfectly possible and valid for the server tool to change its mind and for `readAccess()` to return `aitFalse` at a later time.

Note that it's entirely possible for the user and the host name to change during a channel's lifetime. Thus, the strings passed to `createChannel()` may not remain accurate throughout a channel's lifetime. When either the user name or the host name changes, the server library will call `setOwner()` which is passed two arguments, a pointer to a string representing the user's new name and a pointer to a string representing the host's new name.

```
virtual void setOwner(const char * const pUserName,
                    const char * const pHostName)
```

`setOwner()` must be redefined to update the user and host name kept track of by the server tool, if it keeps track of the two names at all.

The current Portable Server API also allows a the server tool to post access rights events. Access rights events occur when a client's access to a PV changes. The current client-side Channel Access API allows a client to request that it be notified when its access rights to a channel change, thus establishing a monitor for access rights events on that channel. Thus, if a server tool decides to implement access control, and a client's access to a PV changes throughout the lifetime of the channel, then the server tool should call `postAccessRightsEvent()` when the change occurs. `postAccessRightsEvent()` is a member function of the `casChannel` class which the server tool can use to inform the server library that the access rights for the current channel have changed. The server library will then inform the client.

Example

As an example of access control, here is the class `myChannel`, which is derived from the `casChannel` class and which provides implementations for `readAccess()`, `writeAccess()`, and `setOwner()`. It's easy to understand, and all its functions are defined in the class declaration:

```
class myChannel : public casChannel {
public:
   myChannel (const casCtx &ctx,
             const char * const pUserName,
             const char * const  pHostName) :
      casChannel(ctx)
   {
      User = pUserName;
      Host = pHostName;
   }

   aitBool writeAccess () const
   {
      if (strcmp(User.string(), "John") == 0)
         return aitFalse;
      else return aitTrue;
   }
   void setOwner(const char *const pUserName,const char *const
pHostName)   {
      User = pUserName;
      Host = pHostName;
   }
private:
   aitString User;
   aitString Host;
};
```

The class will keep track of the name of the user and host with the two private members, `User` and `Host`, which are `aitString` objects (see aitTypes.h). The constructor initializes them both, as well as the `casChannel` base class. `setOwner()` resets each of the members. Remember that `setOwner()` is called when the user or host name changes during the time the channel is active.

Our redefinition of `writeAccess()` returns `aitFalse` if ❽john❾is the user; otherwise, it returns `aitTrue`. We don't redefine `readAccess()`. The default of `readAccess()` simply returns `aitTrue` each time it's called. Thus, `myChannel` allows read access to all clients. It allows write access to all clients except when the user's name is ❽john.❾

Chapter 4: Working with Arrays

The current Channel Access client-side API allows the client to make a request to read or write *n* elements of an array, as long as *n* does not exceed the maximum number of elements in the array. However, the client-side API doesn't allow a client's request to specify the starting element of the request. The starting element is always the first element, at index 0. For example, if a client requested to read 5 elements from a 15 element array, the elements read would be elements 0-4.

Therefore, in order for a server tool to be fully compatible with the client-side API, a server tool must provide a way to read 0 through *n*-1 elements of an array, provided that *n* does not exceed the total number of elements in the array. Of course, other server tools may provide more flexibility when dealing with arrays, such as reading the four middle elements of a ten-element array, though there is currently no way for a Channel Access client to make such a request. This chapter will show how to manipulate arrays using the `gddAtomic` class.

1. The gddAtomic Class

The `gdd` library provides a class, `gddAtomic`, as a container for arrays. `gddAtomic` is derived from the `gdd` class. It provides a way to define the size of an array of any dimensions. As mentioned in the first chapter, a `gdd` object contains a union member which itself contains members of all the architecture-independent types as well as pointer to `void`. For `gddAtomic` objects, the `void` pointer should point to the array. The architecture-independent type of the array can be retrieved by calling the `gdd::primitiveType()` which returns an enumerated type, `aitEnum`. The `aitEnum` type has enumerators for all the architecture-independent types, `aitEnumInt8` for an eight-byte integer type, for example, as well as an enumerator for unknown types, `aitEnumInvalid`.

Defining a gddAtomic Object

As an example, let's create a small, two dimensional array as part of a `gddAtomic` object, give it some values, and then write those values to standard output.

Firstly, a `gddAtomic` array's size is defined by its dimensions as well as the size of those dimensions. Using a `gddAtomic` class, an array of one, two, three, four, or more dimensions can be created. There are several ways to initialize a `gddAtomic` object. One way, which we won't discuss here, is to simply pass the `gddAtomic` constructor another `gddAtomic` object, in which case the `gddAtomic` object will be initialized with all the characteristics of the argument as well as its data. Another way is to pass the `gddAtomic` constructor an integer representing an application type and an `aitEnum` type representing the architecture-independent type that the array will hold. Or a `gddAtomic` object can be initialized with only an application type. Each of these two methods, of course, doesn't define the array's bounds, but is provided so that the bounds can be defined later. Here are the prototypes for these three constructors:

```
gddAtomic(gddAtomic* ad);
gddAtomic(int app);
gddAtomic(int app, aitEnum prim);
```

There are two other constructors that let you define a `gddAtomic` object's array size when you create the object. For these, in addition to the object's application type and primitive type, you must provide the dimensions of the array as well as the size of each dimension called the *bounds* of the dimension. There are two ways to do this. For the first, pass the constructor an application type (`int`), a primitive type (`aitEnum`), the dimensions of the array (`int`), and an array of type `aitUint32`. The number of elements in the last argument must have at least as many elements as dimensions in the array. For example, for a three-dimensional array, the last argument must contain at least three elements; any additional elements are ignored. Each element represents the size of each dimension, starting with the first dimension.

As an alternative way of creating a `gddAtomic` object, you can pass, instead of an array, integers as additional arguments, each of which represents the size of the array's dimensions. So, instead of an array of three elements, you would pass three additional arguments to the constructor. For example, to initialize a three-dimensional array of 5 columns, 4 rows, and 2 planes, we would initialize the `gddAtomic` object like so:

```
#include <gdd.h>
int main()
{
   gddAtomic Array(1, aitEnumInt8, 3, 5, 4, 2);
   return (0);
}
```

Here are the prototypes for these two `gddAtomic` constructors:

```
gddAtomic(int app, aitEnum prim, int dimen, aitUint32*
size_array);
gddAtomic(int app, aitEnum prim, int dimen, ...);
```

A `gddAtomic` object is defined by its dimensionsÄi.e., whether it's a one-, two-, or three-dimensional arrayÄand by the bounds of each dimension. Each dimension has its bounds. The array's bounds are the index of the first element and the number of elements in the array. By default, when you initialize a `gddAtomic` object and specify the bounds, the index of the first element is 0, but can be adjusted to be another positive integer. Thus, if we initialized a two-dimensional array like so,

```
gddAtomic Array(1, aitEnumInt8, 2, 4, 3);
```

the result would be an array where the first dimension has the bounds 0 and 4, and the second dimension has the bounds 0 and 3. This is the equivalent of creating a conventional C array as in,

```
aitInt8 Array[3][4];
```

The bounds of a `gddAtomic` object can be retrieved using the `getBound()` function, whose prototype is:

```
gddStatus getBound(unsigned dim_to_get, aitIndex& first,
                   aitIndex& count);
```

For the first argument, pass a positive integer that specifies the dimensions whose bounds you wish to retrieve; the second and third arguments must be lvalues of type `aitIndex`. These last two arguments are passed by reference. The function will write the index number of the first element to the second argument, and the element count to the third argument. Thus, for an array initialized as

```
gddAtomic Array(1, aitEnumInt8, 2, 4, 3);
```

the following call to `getBound()` will write 0 into `First` and 3 into `Nelem`:

```
aitIndex First;
aitIndex Nelem;
Array.getBound(2, First, Nelem);
```

because the index of the first element by default is 0 and the size of the second dimension is 3. You can also change the bounds of an object using the `setBound()` function. It also accepts three arguments, the first being the dimension for which you want the bounds changed, the second being the new value of the first element index, and the third argument being the new element count. Using the `setBound()` function, you cannot, however, increase or decrease the array's original dimensions. Thus, to adjust the bounds of a two-dimensional array so that the first dimension consists of 10 elements and its first element is 1, you would make the following call:

```
Array.setBound(1, 1, 10);
```

Accessing Arrays

The `gddAtomic` class does nothing more than hold the actual array and information about it. It doesn't allocate memory for the actual array. The server library uses it because the server library treats all data as `gdd` objects, `gddAtomic` being a derived class of the `gdd` class.

Destroying gddAtomic objects differs somewhat from gddScalar objects. Firstly, all gdd objects should be put on the "free store", i.e., they should be created using the new operator or otherwise dynamically allocated. When the reference count of the object decrements to zero, the gdd library will destroy a gdd object. Since a gddAtomic object may reference a dynamically allocated array, this array should also be destroyed when the gddAtomic container is destroyed. The way the library accomplishes this is to call the run() function, which is a virtual function of the gddDestructor class. The library passes the pointer that references the array in the gddAtomic object to run(). The argument to run() is a pointer to void, void*. Thus, to deallocate a dynamically allocated array referenced by a gddAtomic object, a program should derive a class from gddDestructor and redefine the run() virtual function. This redefinition of run() should cast the void pointer to the appropriate type and then deallocate it.

As mentioned previously, a gdd object contains a union member which has members for each of the architecture-independent types, as well as a pointer to void. This pointer is used for arrays. It can be any valid type, though it should match the architecture-independent type indicated by the primitive type (aitEnum primitiveType()). This pointer can be retrieved using the dataPointer() function. After the pointer is retrieved, the array can be manipulated like a normal array. Another function that may be of use when dealing with arrays is the putRef() function. By passing an array pointer to this function, you can set the void pointer to point to the new array. In addition, the second argument to putRef() can be a pointer to a gddDestructor object or an object derived from gddDestructor. Thus, as well as passing a reference to an array, a program can pass a reference to a gddDestructor object that will take care of the deallocation of the array.

The = operator is overloaded for the gddAtomic class. The operator=() function is invoked whenever a gddAtomic object appears to the right of the assignment operator and a pointer to a valid architecture-independent type appears to the left of the operator. The pointer is then set to point to the array contained in the gddAtomic object. It essentially does the same thing as dataPointer().

Here is a small program that creates an array, uses the putRef() function to install it in a gddAtomic object, and then prints out the elements of the array. In addition, it derives a class from gddDestructor to destroy the dynamically allocated array referenced by the gddAtomic object:

```
#include <gdd.h>

class myDestructor : public gddDestructor {
public:
    myDestructor() : gddDestructor() { }
    void run(void*);
}
```

```
void myDestructor::run(void* v)
{
   aitInt16* p16= (aitInt16*)v;
   delete [] p16;
}

int main()
{
   gddAtomic *pGdd;
   aitUint16 *pArray;
   pGdd = new gddAtomic(1, aitEnumUint16, 1, 10);
   pArray = new aitUint8[10];
   aitIndex i, n;
   for(int i = 0; i < 10; i++)
      pArray[i] = (aitUint8)i;
   pGdd->putRef(pArray, new myDest);
   pGdd->getBound(1, i, n);
   for(;i<n-1;i++)
      cout<<*(pGdd->dataPointer()+i)<<endl;
return 0;
}
```

It's important to note that when the `primitiveType()` function of a `gdd` object returns `aitEnumString`, this does not mean that the void pointer points to a character array. Instead, `aitString` is a separate type. Thus, an array of type `aitString` would actually be an array of `aitString` objects and thus an array of strings, instead of an array of type `char`.

Using Arrays with the Server Tool

As mentioned above, existing Channel Access clients can make requests to read or write multiple elements of an array channel. However, the current client-side API doesn't allow a client to request the beginning index of operation. For instance, a client cannot make a request to change elements 11-16 of a 20-element array. A client can make a request to change elements 0-16, or 0-10, or 0-*n*-1, as long as *n* does not exceed the total number of elements of the array.

There are basically two possible ways for a server tool to deal with arrays. The first and easiest way does not fully support current CA client requests because it will not read *n* elements from an array or write to *n* elements of an array. It simply references the array.

An example of this method was used in the last chapter. The server library calls `casPV::write()` when a client makes a write or put request for a PV. The second argument passed to `write()` is a `gdd` object which is passed by reference. This `gdd` object can be either a `gddScalar` object or a `gddAtomic` object, but for `write()` it cannot be a `gddContainer` object. Whichever `gdd` class it is, it is the value sent by the client, the value to which the client wants the PV changed. For arrays or scalar objects, it's possible for the server tool to simply reference the object; that is, set a pointer to point to it, and then call `reference()` for the object. At the next call to `write()`, the old `gdd` object should be unreferenced by a call to `unreference()` and the new value referenced. Here is an example of a `write()` function which does this where `pVal` is simply a pointer to a `gdd` object that contains the PV's current value:

```
caStatus myPV::write(const casCtx &ctx, gdd &value)
{
    if (pVal)
        pVal->unreference();
    pVal = &Value;
    pVal->reference();
    return S_casApp_success;
}
```

When called, the `reference()` function increments a counter. The `unreference()` function decrements the same counter. If referenced properly, the `gdd` object's counter should indicate the number of pointers that are referencing it. When the counter becomes zero, it is destroyed.

By simply referencing the `gdd` object, a server tool doesn't have to duplicate any information, and hence the amount of memory used is minimal. This is especially desirable when dealing with large arrays.

The other way to handle arrays is to copy the information from them or the requested part of them. Using this method, a server tool will fully support the current Channel Access client-side API since the API allows requests to read from or write to *n* elements of a PV array, where *n*-1 may not be the last element in the array. Therefore, a client can validly request to read 0-*n* elements of a 20-element array, as long as *n* does not exceed the index of the last element, 19 in this case.

Here is an example of an implementation that can write *n* elements from the array passed to it where `pVal` is pointer to a `gdd` object that contains the PV's current value. For simplicity's sake, assume that only arrays are going to be written to the PV.

```
caStatus myPV::write(const casCtx &ctx, gdd &value)
{
    aitIndex first1, first2, nelem1, nelem2;
    unsigned dim1, dim2;
    dim1 = value.dimension();
    dim2 = pVal->dimension();
    if(dim2>1 || dim1>1)
        return S_casApp_outOfBounds;
    else {
        value.getBound(0, first1, nelem1);
        pVal->getBound(0, first2, nelem2);
    if (first1 !=0 || first2 != 0)
        return S_casApp_outOfBounds;
    if (nelem2<nelem1)
        return S_casApp_outOfBounds;
    else
```

```
       aitConvert(pVal->primitiveType(),  // dest. type
              pVal,                                    //
destination
              value.primitiveType(), // src. type
              &value,                                        //
source
              nelem2 );                                      //
count
   }
   return S_casApp_success;
}
```

Our function is very simple. It retrieves the dimensions for each `gddAtomic` object, and if either object's dimensions are greater than zero, it returns `S_casApp_outOfBounds`. Currently, the client-side API won't allow clients to make requests for multi-dimensional arrays, so our write function won't support writing to arrays of more than one dimension. Note, however, that at certain EPICS sites, multi-dimensional arrays have been used in requests by passing one wrapped by a single-dimensional array. Whether a server tool wishes to provide the capability to deal with arrays should depend on the site.

Next `getBound()` is called. The zero as the first argument tells `getBound()` to write the index of the first element and the count of the array from the first dimension into the second and third arguments. Remember that dimensions are indexed as they would be in any normal array, that is 0 to *n*-1 where *n* is the number of dimensions.

Next, the first element count of each array object is checked; if it's not zero, `S_casApp_outOfBounds` is returned. Thus, our `write()` function does not support write operations where the first index is something other than zero, which are not necessary as the existing client API doesn't support such operations, though this may change in a future release.

Then the element counts of the objects are compared. If the destination object, the `gddAtomic` object to be changed, has an element count less than the element count of the source object, the error code `S_casApp_outOfBounds` is returned once again. Finally, `aitConvert()` is called. `aitConvert()` is a function provided as part of the `gdd` library. It copies the data from an array of one architecture-independent type to an array of another architecture-independent type, converting the data to the type of the destination array. It will also convert scalar values. Actually, `aitConvert()` merely calls the appropriate function from an array of functions that contains all the functions necessary to convert from one architecture-independent type to another. The limitation of `aitConvert()` is that only one-dimensional arrays are supported. Its prototype is as follows

```
void aitConvert(aitEnum desttype, void* dest, aitEnum srctype,
                const void* src, aitIndex count);
```

The first argument is the primitive type of the destination array, the type to be converted to. The second argument is the address of the destination array. The third argument is the primitive type of the source array, the type to be converted from, the fourth argument is the address of the source array, and the fifth argument is the number of elements to be copied. As the fifth argument, we passed the number of elements of the source array, which is the number of elements that the client has requested to be written to the source array. If the destination array's element count is less than the source array's, the error code `S_casApp_success` is returned to the server library.

Our `write()` function will support all client requests for array ❽put❾operations. To support more flexibility in dealing with arrays is not difficult. For instance, to implement a function to deal with multi-dimensional arrays, one would simply have to index the array as a series of one-dimensional arrays, passing the sub-array to `aitConvert()` instead of the whole array. Because `aitConvert()` automatically starts indexing both the destination and source arrays at 0, copying chunks out of one array into another presents a somewhat more difficult problem. For example, if you wanted to copy elements 4-10 of a 15-elements array, you would have to do this by indexing each element and calling `aitConvert()`.

In any case, the documentation for the gdd Library should be consulted.

Appendix A: myServer

This is the full listing of the program presented in *Chapter 2: Getting Started*. The actual executable program is called `myServer`. It's related source files are myServer.h, which contains all the class declarations, Server.cc, which contains `main()`, myServer.cc, which contains all the function definitions for the myServer class, and myPV.cc, which contains all the function definitions for the `myPV` class.

To compile `myServer`, compile object files for myPV.cc, myServer.cc, and Server.cc. Then link them along with the libraries libcas.a, libca.a, libCom.a, and libgdd.a. The libraries should be linked in a that order:

libcas.a

libca.a

libCom.a

libgdd.a

Another order may cause compiler errors. The program has been compiled to run on a Sun Sparc workstation.

This version of `myServer` doesn't implement asynchronous IO or access rights. In Appendix B is a listing of a different version of `myServer` that does, though most of the code is the same.

```
/*** myServer.h: Contains class declarations. ***/
#include <casdef.h>
#include <osiTimer.h>
#include <gddApps.h>
#include <gddAppFuncTable.h>

class myPV;
class pvAttr;

      // The server class. Provides the pvExistTest() and createPV()
// functions as well as the function table
class myServer : public caServer{
 public:
  myServer(unsigned pvCountEstimate);
  pvExistReturn pvExistTest (const casCtx &ctx, const char
*pPVName);
  static const pvAttr *findPV(const char* Name);
  pvCreateReturn createPV(const casCtx &ctx, const char *pPVName);
  static gddAppFuncTableStatus read(myPV &pv, gdd &value)
  {
   return myServer::funcTable.read(pv, value);
  }
 private:
  static const pvAttr pvList[];
  static gddAppFuncTable<myPV> funcTable;
};

// The Process Variable class, myPV. Provides the read() and
write()
// functions. The read() function calls the myServer::read()
function
// which calls the appropriate function or functions from its
function
// table. The actual functions to becalled are members of
// the myPV class, readStatus() to readUnits().
class myPV : public casPV{
 public:
  myPV (const caServer &cas, const pvAttr &attributes);
  caStatus interestRegister()
  {
   interest = aitTrue;
   return S_casApp_success;
  }
  void interestDelete() { interest = aitFalse; }
  caStatus beginTransaction();
  void endTransaction();
  caStatus read(const casCtx &ctx, gdd &prototype);
  caStatus write(const casCtx &ctx, gdd &value);
  aitEnum bestExternalType();
  gddAppFuncTableStatus readStatus(gdd &value);
  gddAppFuncTableStatus readSeverity(gdd &value);
  gddAppFuncTableStatus readPrecision(gdd &value);
  gddAppFuncTableStatus readHopr(gdd &value);
  gddAppFuncTableStatus readLopr(gdd &value);
  gddAppFuncTableStatus readHighAlarm(gdd &value);
  gddAppFuncTableStatus readHighWarn(gdd &value);
  gddAppFuncTableStatus readLowWarn(gdd &value);
  gddAppFuncTableStatus readLowAlarm(gdd &value);
  gddAppFuncTableStatus readHighCtrl(gdd &value);
  gddAppFuncTableStatus readLowCtrl(gdd &value);
  gddAppFuncTableStatus readValue(gdd &value);
  gddAppFuncTableStatus readUnits(gdd &value);
 private:
  static int currentOps;
  const pvAttr& attr;
  aitBool interest;
};

// This class is not part of the server interface, but is only
used
// here in order to keep track of PV values. It contains members
for
// storing and accessing all the application types needed to
satisfy a
// DBR_CTRL request. DBR_STS requests can also be satisfied. Since
DM
```

```
// makes DBR_CTRL requests for each controller or monitor
initially,
// and then makes DBR_STS requests subsequently,this class
contains
// the attributes needed to work with DM clients.
class pvAttr {
 public:
  pvAttr (const char *pName) : name(pName)
    {
      pValue = new gddScaler(gddAppType_value, aitEnumFloat64);
      hopr = 100, lopr = 0;
      units = "Jolts";
      high_alarm = 101, high_warning = 95;
      low_warning = 5, low_alarm = -1;
      high_ctrl_limit = 100, low_ctrl_limit = 0;
      precision = 4;
    }

  const aitString &getName () const { return name; }
  double getHopr () const { return this->hopr; }
  double getLopr ()  const { return this->lopr; }
  double getHighAlarm () const { return this->high_alarm; }
  double getHighWarning () const { return this->high_warning; }
  double getLowWarning () const { return this->low_warning; }
  double getLowAlarm () const { return this->low_alarm; }
  double getHighCtrl () const { return this->high_ctrl_limit; }
  double getLowCtrl ()const  { return this->low_ctrl_limit; }
  short getPrec () const { return this->precision; }
  gdd* getVal () const { return this->pValue; }
  aitString getUnits () const { return this->units; }
 private:
  const aitString name;
  double hopr;
  double lopr;
  aitString units;
  double high_alarm;
  double high_warning;
  double low_warning;
  double low_alarm;
  double high_ctrl_limit;
  double low_ctrl_limit;
  short precision;
  gdd *pValue;
};


/*** myServer.cc: contains function for myServer class. ***/
#include "myServer.h"
// Here the myServer::pvList[] static member is initialized with
two
// PVs called ProcessVariable1 and ProcessVariable2.
const pvAttr myServer::pvList[] = {
  pvAttr("ProcessVariable1"),
  pvAttr("ProcessVariable2")};
```

```
// Constructor for myServer. After passing arguments to caServer
// constructor all of the read functions are installed in the
function
// table.
 myServer::myServer(unsigned pvCountEstimate) :
 caServer(pvCountEstimate)
{
   funcTable.installReadFunc("status", myPV::readStatus);
  funcTable.installReadFunc("severity", myPV::readSeverity);
  funcTable.installReadFunc("precision", myPV::readPrecision);
  funcTable.installReadFunc("alarmHigh", myPV::readHighAlarm);
  funcTable.installReadFunc("alarmHighWarning",
myPV::readHighWarn);
  funcTable.installReadFunc("alarmLowWarning", myPV::readLowWarn);
  funcTable.installReadFunc("alarmLow", myPV::readLowAlarm);
  funcTable.installReadFunc("value", myPV::readValue);
  funcTable.installReadFunc("graphicHigh", myPV::readHopr);
  funcTable.installReadFunc("graphicLow", myPV::readLopr);
  funcTable.installReadFunc("controlHigh", myPV::readHighCtrl);
  funcTable.installReadFunc("controlLow", myPV::readLowCtrl);
  funcTable.installReadFunc("units", myPV::readUnits);
}

pvExistReturn myServer::pvExistTest(const casCtx &ctx,   //CA
Context
                                                      const char
*pPVName, // PV name
{
  const pvAttr *pPVAttr;
  // If the PV exists, write its name to the canonical PV name
object.
  pPVAttr = myServer::findPV(pPVName);
  if (pPVAttr)
    return pverExistsHere;
  else
    return pverDoesNotExistHere;
}

const pvAttr *myServer::findPV(const char *pName)
{
  const pvAttr *pPVAttr;
  int i;
  short nelem = NELEMENTS(myServer::pvList);

  for(pPVAttr = myServer::pvList, i = 0; i < nelem; i++,
pPVAttr++){
    if (strcmp(pName, pPVAttr->getName().string()) == 0)
      return pPVAttr;
  }
  return NULL;
}

pvCreateReturn myServer::createPV(const casCtx &ctx, const char
*pPVName)
{
  const pvAttr *pAttr;
  myPV *pPV;
  // If PV doesn't exist, return NULL. Otherwise, return a pointer
  // to a new myPV object.
  pAttr = myServer::findPV(pPVName);
  if (!pAttr)
    return S_casApp_pvNotFound;
  pPV = new myPV(*this, *pAttr);
  if(pPV)
   return S_casApp_noMemory;
  else
   return pPV
}
/*** myPV.cc: contains functions for myPV class ***/
#include "myServer.h"
#include <iostream.h>

 myPV::myPV(const caServer &cas, const pvAttr &attributes) :
  attr(attributes),
  casPV(cas),
  interest(aitFalse)
{
  double value;                    // The initial value of the PV.
  gdd *pValue =  attr.getVal();    // Get pointer to gdd object.
  if(!pValue)
    return;

  // rand() is used to generate a random number from 0 to
RAND_MAX.
  // This number is made to fit in the 0 - 100 range.
  value = (double)rand();
  while (value < 100.0)
    value/=100.0;

  // Use the gdd::putConvert() function to put the value in the
gdd
  // object. Then use the gdd::setStat() and gdd::setSevr() to set
  // the appropriate status and serverity for the PV.
  pValue->putConvert(value);
  if (value >= 95){
    pValue->setStat(epicsAlarmHigh);
    pValue->setSevr(epicsSevMinor);
  }
  else if (value <=5 ){
    pValue->setStat(epicsAlarmLow);
    pValue->setSevr(epicsSevMinor);
  }
  else {
    pValue->setStat(epicsAlarmNone);
    pValue->setSevr(epicsSevNone);
  }
}

caStatus myPV::beginTransaction()
{
  // Trivial implementation that informs the user of the number of
  // current IO operations in progress for the server tool.
currentOps
  //  is a static member.
  currentOps++;
  cerr << "Number of current operations = " << currentOps << "\n";
  return S_casApp_success;
}

void myPV::endTransaction()
{
  currentOps--;
}

caStatus myPV::read(const casCtx &ctx, gdd &prototype)
{
  // Calls myServer::read() which calls the appropriate function
  // from the application table.
  return myServer::read(*this, prototype);
}

gddAppFuncTableStatus myPV::readStatus(gdd &value)
{
  gdd *pValue = attr.getVal();
  if(pValue)
    value.putConvert(pValue->getStat());
  else
    value.putConvert(epicsAlarmUDF);
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readSeverity(gdd &value)
{
  gdd *pValue = attr.getVal();
  if(pValue)
    value.putConvert(pValue->getSevr());
  else
    value.putConvert(epicsSevNone);
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readPrecision(gdd &value)
{
  value.putConvert(attr.getPrec());
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readHopr(gdd &value)
{
  value.putConvert(attr.getHopr());
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readLopr(gdd &value)
{
  value.putConvert(attr.getLopr());
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readHighAlarm(gdd &value)
{
  value.putConvert(attr.getHighAlarm());
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readHighWarn(gdd &value)
{
  value.putConvert(attr.getHighWarning());
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readLowWarn(gdd &value)
{
  value.putConvert(attr.getLowWarning());
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readLowAlarm(gdd &value)
{
  value.putConvert(attr.getLowAlarm());
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readHighCtrl(gdd &value)
{
  value.putConvert(attr.getHighCtrl());
  return S_casApp_success;
}
gddAppFuncTableStatus myPV::readLowCtrl(gdd &value)
{
  value.putConvert(attr.getLowCtrl());
  return S_casApp_success;
}

gddAppFuncTableStatus myPV::readValue(gdd &value)
{
  // If pvAttr::pValue exists, then use the gdd::get() function to
  // assign the current value of pValue to currentVal; then use
the
  // gdd::putConvert() to write the value into value.
  gdd *pValue = attr.getVal();
  double currentVal;
  if(!pValue)
    return S_casApp_undefined;
  else {
    pValue->getConvert(currentVal);
    value.putConvert(currentVal);
    return S_casApp_success;
  }
}
```

```
gddAppFuncTableStatus myPV::readUnits(gdd &value)
{
  value.put(attr.getUnits());
  return S_casApp_success;
}

// bestExternalType() is a virtual function that can redefined to
// return the best type with which to access the PV. Called by the
// server library to respond to client request for the best type.
aitEnum myPV::bestExternalType()
{
  gdd* pValue = attr.getVal();
  if(!pValue)
    return aitEnumInvalid;
  else
    return pValue->primitiveType();
}

caStatus myPV::write(const casCtx &ctx, gdd &value)
{
  struct timespec t;
  osiTime current(osiTime::getCurrent());
  gdd *pValue;
  caServer *pServer = this->getCAS();
  double newVal;

  // Doesn't support writing to arrays or container objects
  // (gddAtomic or gddContainer).
  if(!(value.isScalar()) || !pServer)
    return S_casApp_noSupport;

  pValue = attr.getVal();
  // If pValue exists, unreference it, set the pointer to the new
gdd
  // object, and reference it.
  if(pValue)
    pValue->unreference();
  pValue = &value;
  pValue->reference();

  // Set the timespec structure to the current time stamp the gdd.
  current.get(t.tv_sec, t.tv_nsec);
  pValue->setTimeStamp(&t);

  // Get the new value and set the severity and status according
  // to its value.
  value.get(newVal);
  if (newVal > 100){
    value.setStat(epicsAlarmHiHi);
    value.setSevr(epicsSevMajor);
  }
  else if (newVal >= 95){
    value.setStat(epicsAlarmHigh);
    value.setSevr(epicsSevMinor);
  }
  else if (newVal <=5 ){
    value.setStat(epicsAlarmLow);
    value.setSevr(epicsSevMinor);
  }
  else if (newVal < 0){
    value.setStat(epicsAlarmLoLo);
    value.setSevr(epicsSevMajor);
  }
  if(interest == aitTrue){
    casEventMask select(pServer->valueEventMask |
                        pServer->alarmEventMask);
    postEvent(select, *pValue);
  }
  return S_casApp_success;
}

/*** Server.cc: contains main(). ***/
#include <fdMgr.h>
#include "myServer.h"

// These static members must be re-declared in this file.
gddAppFuncTable<myPV> myServer::funcTable;
int myPV::currentOps;

main()
{
  myServer *pCAS;
  // Create server object.
  pCAS = new myServer(5u);
  if(!pCAS)
    return;
  pCAS->setDebugLevel(5u);

  // Loop forever
  osiTime delay(1000u, 0u);
  while (aitTrue)
    // fileDescriptorManager is a predeclared object found in
fdMgr.h
    fileDescriptorManager.process(delay);
}
```

Appendix B: MyAsyncWriteIO, MyAsyncReadIO, and myChannel Classes

This a listing which presents a trivial implementation of the `casChannel` and asynchronous IO completion. The derived classes are called `myChannel`, `myAsyncReadIO`, and `myAsyncWriteIO`. Because most of the code is the same as that presented in Appendix A, only the declarations of the `MyAsyncWriteIO`, `MyAsyncReadIO`, and `myChannel` classes are listed here, in addition to the implementations for the `myPV::read()` and `myPV::write()` functions.

```
class myChannel : public casChannel {
 public:
  myChannel(const casCtx &ctx, const char * const pUserName,
       const char * const  pHostName) : casChannel(ctx)
  { User = pUserName; Host = pHostName; }
  aitBool readAccess () const { return aitTrue; }
  aitBool writeAccess () const
  {
      if (strcmp(User.string(), "John") == 0)
       return aitFalse;
      else
       return aitTrue;
  }
  aitBool confirmationRequested () const { return aitTrue; }
  void setOwner(const char *const pUserName,
              const char *const pHostName)
  {
      User = pUserName;
      Host = pHostName;
  }
 private:
  aitString User;
  aitString Host;
};
```

```
class myAsyncReadIO : public casAsyncReadIO, public osiTimer {
 public:
  myAsyncReadIO(const casCtx &ctx, gdd &Value, myPV &pv) :
    casAsyncReadIO(ctx), PV(pv), osiTimer(osiTime(10.0))
    {
      pValue = &Value;
      pValue->reference();
    }
  void expire();
  gdd *pValue;
 private:
  myPV &PV;
};

class myAsyncWriteIO : public casAsyncWriteIO, public osiTimer {
 public:
  myAsyncWriteIO(const casCtx &ctx, gdd &Value, myPV &pv) :
    casAsyncWriteIO(ctx), PV(pv), osiTimer(osiTime(10.0))
    {
      pValue = &Value;
      pValue->reference();
    }
  void expire();
  gdd *pValue;
 private:
  myPV &PV;
};
```

/*** myAsyncIO.cc: contains function definitions for expire(). ***/

```
#include "myServer.h"
#include <iostream.h>

void myAsyncReadIO::expire()
{
  caStatus status, status1;
  status = myServer::read(PV, *pValue);
  status1 = postIOCompletion(status, *pValue);
  if (status1 != S_casApp_success)
    cerr <<"Error returned by postIOCompletion: "
         <<"myAsyncReadIO::expire()."<<endl;
}

void myAsyncWriteIO::expire()
{
  gdd *pValue1;
  caStatus status;
  pValue1 = PV.getAttr().getVal();
  // Just reference the darn thing!
  if(pValue1)
    pValue1->unreference();
  pValue1 = pValue;
  status = postIOCompletion(S_casApp_success);
  if (status != S_casApp_success)
    cerr <<"Error returned by postIOCompletion: "
         <<"myAsyncReadIO::expire()."<<endl;
}
```

/*** myPV.cc: contains redefintions of read() and write() ***/

```
caStatus myPV::read(const casCtx &ctx, gdd &prototype)
{
  myAsyncReadIO *pIO;
  pIO = new myAsyncReadIO(ctx, prototype, *this);
  if (!pIO)
    return S_casApp_noMemory;
  else
    return S_casApp_asyncCompletion;
}

caStatus myPV::write(const casCtx &ctx, gdd &value)
{
  myAsyncWriteIO *pIO;
  pIO = new myAsyncWriteIO(ctx, value, *this);
  if (!pIO)
    return S_casApp_noMemory;
  else
    return S_casApp_asyncCompletion;
}
```