CDEV: An Object-Oriented Class Library for Developing Device Control Applications

Jie Chen and Graham Heyes
Data Acquisition Group

Walt Akers, Danjin Wu and William A. Watson III
Control Software Group

Continuous Electron Beam Accelerator Facility
12000, Jefferson Avenue
Newport News, VA 23185

Abstract

The Control Device API (CDEV) is a highly modular and extensible object-oriented C++ class library that provides a standard interface to one or more underlying control or data acquisition packages through a common framework into which system developers can fit custom code. It defines a set of abstract classes from which a new CDEV service-layer can be developed by inheritance and accessed with the same API through run time dynamic binding. All I/O in the system is handled as synchronous or asynchronous messages to devices that may span multiple services. CDEV routes messages to appropriate services by a name service and dispatches multiple services to handle service specific I/O events. In addition, CDEV handles data transfer through a data object that may contain multiple tagged values of different types, allowing flexible I/O between clients and servers. This paper presents the design, implementation and current status of CDEV, and shows that CDEV can be a starting point to achieve the goal of sharing software in control system applications.

1 Motivation

In building a control or a data acquisition (DAQ) application, programmers usually have to conform to application program interfaces (API) provided by one or more underlying control or DAQ services such as EPICS [1] and CODA [2]. Developing a control/DAQ application is difficult since it requires detailed knowledge of the control/DAQ services such as network connection establishment, creation and synchronization of all I/O requests and data conversion handling. As control systems become more complex, the applications tend to use several services and several APIs developed at different sites at the same time. Any applications using multiple APIs will suffers from the following limitations:

- Steep Learning Curve: Many control/DAQ services have different purposes and thus have different and complicated APIs. This requires a significant effort to learn and use. For example, the channel access [3] interface has its own set of I/O operations (eg. ca_array_get (), ca_array_get_callback()), while CODA has daReadInt(), daWriteInt() and so on.
- Poor Portability: It is difficult to take a useful application from one site and API to a different site and API. On the other hand, APIs that use object-oriented features such as inheritance and dynamic binding are typically easy to port and to extend transparently [4].
- Maintenance Difficulty: It is very difficult for an application to accommodate any changes in APIs of existing services. This increases the complexity of maintaining application source code. Any new changes of API of an existing service will result in major modification of the applications.
- Event Handling Loop Dilemma: Usually all the services have their own event handling loop. This makes it difficult to decide which event loop to use in an application and how to dispatch and synchronize all I/O events from all services.

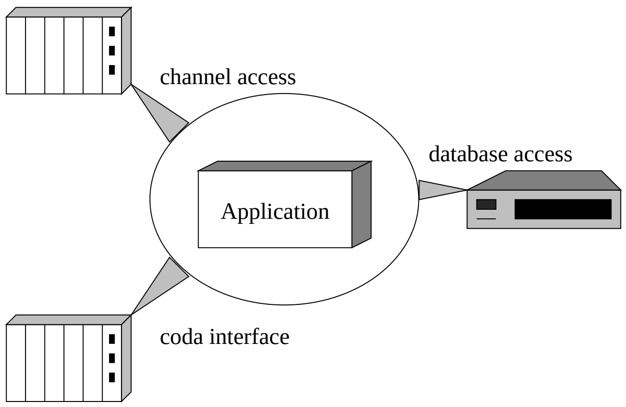

Figure 1: A typical application with multiple services

Consider a typical application shown in Figure 1. The application has to stop or start a data acquisition run according to hardware information retrieved from the channel access service and store some important data into a data base. The application not only has to handle requests to/from different services, but also has to coordinate all requests, such that no service will be blocked for a unreasonably long period of time. Furthermore, the application has to be modified or rewritten if any of the APIs or services have been changed or the application needs to access a different service.

2 C++ Solution: CDEV

It is impractical to imagine a consistently designed and all- purpose control/DAQ service which fits all the requirements of different sites. A more realistic alternative is to develop an object-oriented interface that not only encapsulates existing services, but also allows new services to be added. Due to the efficiency and wide availability of C++, it make sense to define a simple public interface for all services within inheritance hierarchies and a set of abstract C++ classes from which a new set of classes can be derived and wrapped around the API of a service. In this way, applications can use the same public interface to access different services through the derived C++ classes. We call this C++ class library CDEV.

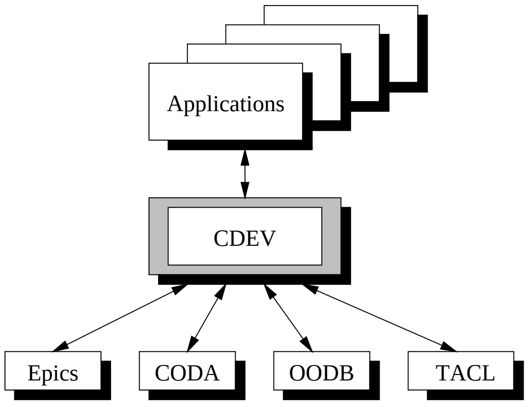

Figure 2: Architectural overview of CDEV

Figure 2 illustrates how the CDEV layer interacts with user applications and multiple services. In this approach, applications use CDEV to access these services via type-secure, object-oriented class interfaces, rather than through a set of service specific APIs. In general, CDEV is designed to reduce the complexity of control/DAQ software development without compromising performance. For example, the CDEV uses C++ language features (such as inline functions) that minimize the performance overhead of the additional layer in some of the critical sections of the CDEV code.

3 Object-Oriented Design and Implementation of CDEV

Throughout CDEV development rigorous object-oriented design is employed to ensure better quality and easier ways to communicate within the development team. This section discusses several topics related to object-oriented design and implementation of CDEV with the help of OMT (Object Modeling Technique) [5] and Booch [6] notation. To begin with, CDEV views an application as a system of *cdevSystem* which keeps information about devices, services and related resources. The system also behaves as a memory manager for all CDEV objects so that applications need not worry about memory management.

To simplify the interface to services, a device (*cdevDevice*) in CDEV is regarded as a named entity which can only respond to a set of messages such as *on*, *off* or *get/set* attributes. All I/O requests in the system are in the form of messages to devices. A message to a device can be handled either synchronously or asynchronously, and is actually carried out by a request object (*cdevRequestObject*) that is created and registered. A device will dispatch messages to a correct request object which in turn sends out to the correct control package or service. Figure 3 shows all forms of I/O requests and object relationships between a device and request objects. This message-based interface simplifies the use of CDEV and also makes it simple to put any message-based language (such as Tcl [7]) on top of CDEV. To allow for the message based interface to work with different services, a new data type *cdevData* is designed to hold different data types with different tags.

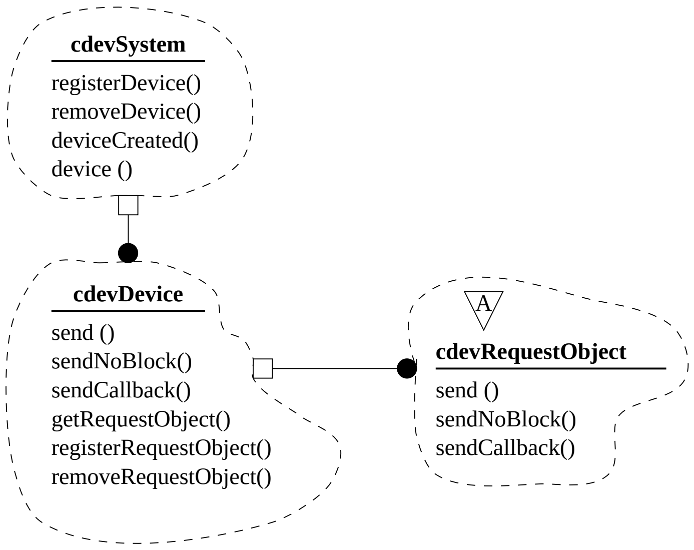

Figure 3: CDEV device and request object class category

In CDEV, a particular control/DAQ service interface is totally separated from applications and also hides the representation and internal structure of all I/O requests to/from a service. Applications using CDEV communicate with packages or services through a service layer which contains a C++ class derived from *cdevService* and a class derived from *cdevRequestObject* which handles all I/O requests. Figure 4 illustrates how CDEV uses an abstract interface for constructing a particular service of *cdevService* and how it constructs an I/O request to the services by a *cdevRequestObject*. This figure also points out the similarity between the design of this part of CDEV and the object-oriented builder pattern [8] which is widely used for a construction process that allows different representations for the object that is constructed. This object-oriented design improves modularity by encapsulating the way a complex object is constructed and represented. Applications need not know anything about the classes that define the *cdevService* or *cdevRequestObject* internal structure; such structures do not appear in the CDEV interface. Only at run-time CDEV passes messages to a name resolution system called *cdevDirectory* (see the following section) to decide which subclass of *cdevService* or which subclass of *cdevRequestObject* to construct. CDEV keeps all constructed objects to enable subsequent calls to get to the objects quickly.

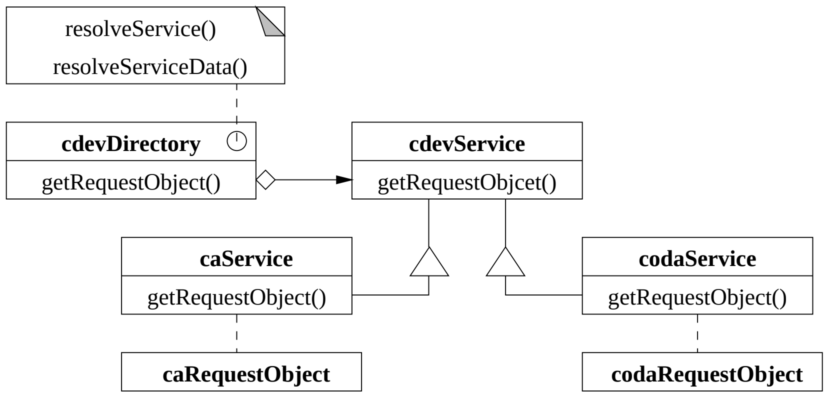

Figure 4: CDEV service and request object creation pattern

In a typical application using multiple services, one or more results due to I/O requests may arrive from different sources (such as channel access ports and CODA communication ports) and it is not desirable to block or continuously poll for incoming I/O events on any individual source. In Figure 5 a system *cdevSystem* maintains a set of services derived from *cdevSerice*. It provides methods of registering and removing these *cdevService* objects from this set at run-time. It also provides an interface for dispatching the appropriate methods of the *cdevService* objects associated with incoming I/O events. The system detects and reports the simultaneous occurrence of different types of events on multiple I/O descriptors given by the *cdevService* objects. When one or more I/O events arrive, the system of *cdevSystem* returns from the event demultiplexing call and dispatches the appropriate method(s) associated with *cdevService* subclass objects registered to handle these events. Therefore CDEV handles concurrent event demultiplexing and dispatching efficiently and frees applications from handling events explicitly.

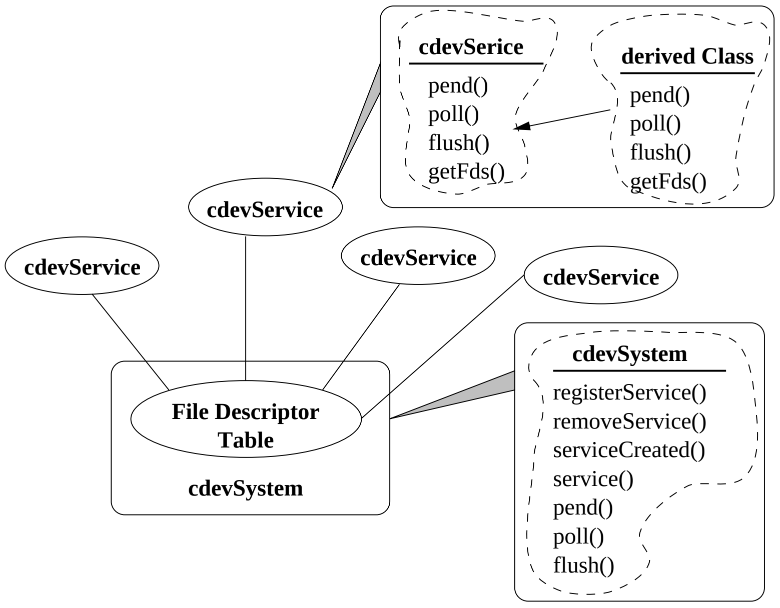

Figure 5: CDEV event dispatching mechanism

CDEV also eliminates the need for complex synchronization of multiple asynchronous I/O requests in applications. Consider the application in Figure 6, which sends out multiple asynchronous I/O requests through several objects derived from *cdevRequestObject* within a *cdevGroup* that maintains a set of *cdevTranObj* objects associated with each request object. The *cdevGroup* provides ways to add and remove these transaction objects from the group at run-time. It also dispatches the appropriate methods for the request objects and removes the associated transaction object when I/O requests have finished with notification callbacks. Therefore a *cdevGroup* object can wait for all I/O requests to finish on a set of I/O request objects. In addition, applications can use *cdevCallback* to notify itself when an asynchronous I/O request has finished.

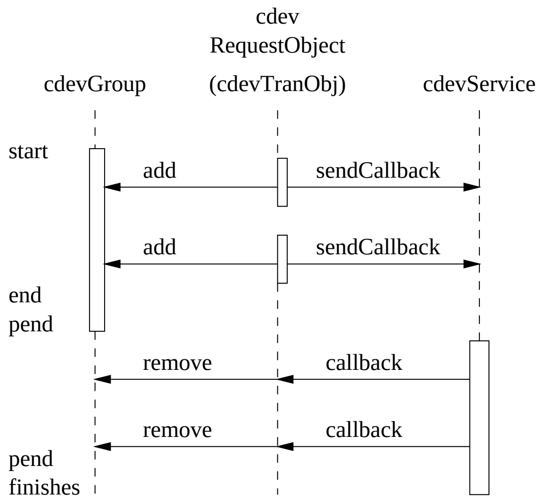

Figure 6: CDEV I/O requests synchronization mechanism

CDEV also allows easy integration of new services. Applications in CDEV only use classes derived from *cdevService* to communicate with services and use I/O request objects derived from *cdevRequestObject* to send out I/O requests in the form of messages. A new service can be integrated into CDEV by developing two new classes. One is derived from *cdevService*, which handles I/O events dispatching. Another is derived from *cdevRequestObject*, which accepts I/O requests in messages and send them out to a service. Once these two classes are implemented and registered into the CDEV name resolution system (see the following section), applications can access the new service without modifying their source code.

4 Related Design and Implementation Issues

Besides the design and implementation issues described in the above section, CDEV also utilizes some common design and implementation techniques to give applications a better interface.

First of all CDEV uses a simple set of enumerated values to denote the status of all CDEV functions. This was chosen rather than using the C++ exception handling mechanism because it will be easier to interface to procedural wrappers for C and Fortran users. CDEV also defines a standard error-reporting class *cdevError* which is inherited by *cdevSystem*. Applications can override the default error reporting function in the *cdevError* class to redirect an error message to an appropriate function.

Secondly, a new data type called *cdevData* is implemented. In order to let applications using CDEV to communicate with different control services or packages, CDEV has to be able to handle different data types in a uniform fashion. Thus a new C++ data type *cdevData* is designed to provide this service. The *cdevData* serves as a repository for data of different types and sizes, accessed by either an integer or character string tag. There is a one-to-one correspondence between integer tags and character tags, so either may be used to insert to retrieve data. Currently a *cdevData* object can hold any type of int, pointer to a character string, char, short, ushort, uint, long, ulong, float, double, or time_val structure and arrays of same. In the future, any structure which is derived from a common parent class can be added or retrieved from *cdevData*. Any applications or new service layers which use *cdevData* for high performance purpose should use integer tags instead of character string tags. Finally, *cdevData* also inlines a lot of its functions to achieve optimal performance.

Thirdly, CDEV contains a name resolution system to locate which service objects derived from *cdevService* to call, and what data to pass to the service in an I/O request. CDEV constructs the name resolution system by parsing an ASCII file that is written by a system programmer in DDL (Device Definition Language) format, and creates a table containing information about devices in a control system. The name resolution system that is called *cdevDirectory* is also derived from *cdevDevice* so that it can be accessed by applications at run-time through the same message based interface. Table 1 summarizes the messages the name resolution system accepts and the results it returns.

Message	Tag Names	Result
service	device, message	service name
serviceData	device, message	data to pass to the service
query	device	list of devices
queryClass	device	parent DDL class
queryAttributes	device, class	all attributes
queryMessages	device, class	all messages
queryVerbs	device, class	all supported verbs
update	value or file	status of the operation
validate	device and class	inheritance relation

Table 1: Supported messages of CDEV name resolution system

Finally, CDEV does not define a new network protocol. It uses the network protocols of its underlying services. For examples, applications using CDEV to access a device via channel access use the channel access protocol.

5 Sample Application

The following code illustrate some of the features of CDEV.

```
#include <cdevSystem.h>
#include <cdevDevice.h>
#include <cdevRequestObject.h>
#include <cdevData.h>
#include <cdevGroup.h>

static void devcallback (int status,
                         void* arg,
                         cdevRequestObject& obj,
                         cdevData& data)
{
    float fval;

    if (data.get (‘‘value’’, &fval) == CDEV_SUCCESS)
         printf (‘‘bdl of %s is %f\n’’, obj.device().name(), fval);
}



main (int argc, char **argv)
{
    cdevSystem& system = cdevSystem::defaultSystem ();
    cdevDevice& dev0 = cdevDevice::attachRef (‘‘magnet0’’);
    cdevDevice& dev1 = cdevDevice::attachRef (‘‘magnet1’’);

    cdevData result0, result1;
    int status = dev0.send (‘‘get bdl’’, 0, result0);
    // do something with result0
    status     = dev1.send (‘‘get bdl’’, 0, result1);
    // do something with result1

    cdevGroup grp;
    grp.start ();
    status = dev0.sendNoBlock (‘‘get current’’, 0, result0);
    status = dev1.sendNoBlock (‘‘get current’’, 0, result1);
    grp.end ();
    grp.pend (4.0);
    // do something with result0 and result1

    cdevCallback callback (devcallback, 0);
    cdevGroup grp;
    grp.start ();
    status = dev0.sendCallback (‘‘monitorOn bdl’’, 0, callback);
    status = dev1.sendCallback (‘‘monitorOn bdl’’, 0, callback);
    grp.end ();
    grp.pend (4.0);

    for (;;)
        system.pend ();
}
```

The main program starts by opening a default *cdevSystem* system that keeps all the information about services and devices. Next, it creates two *cdevDevice*s, dev0 and dev1, with name of magnet0 and magnet1 respectively. Then, two synchronous requests are sent out to the devices and followed by a demonstration of the synchronization method of CDEV using *cdevGroup*. Finally, the program enters a event loop in which the attributes bdl of devices are monitored.

6 Benefits of CDEV interface

The CDEV C++ interface provides several software quality factor improvements:

- Correctness: CDEV improves the type-security of higher level applications which no longer need to access low level C-based APIs of services. Strongly-typed object-oriented CDEV interface detects type errors at compile time rather than at run-time.
- Ease to Use and Easy to Learn: the CDEV interface is organized into a set of C++ classes in a hierarchical structure. Hierarchical APIs are typically easier to learn, since their structure indicates closely related operations. In addition, the message-passing interface of CDEV reduces the learning curve dramatically. Simple applications may be written using only 2 classes: *cdevDevice* and *cdevData*. Providing simpler and compact interface allows application programmers to concentrate on design and implementation, rather than wrestling with all the different sets of low level APIs.
- Easy to Maintain: Applications using CDEV are hidden from all underlying services. All I/O requests sent out by applications are mapped transparently on to appropriate services. Furthermore applications are more immune to any changes of the services and are valid even with a newly introduced service without any modification.
- Extensibility: CDEV offers great extensibility to the applications since it is designed using powerful C++ language features (such as inheritance, dynamic binding and polymorphism). Explicitly, CDEV defines several abstract C++ classes which must be inherited by a service that wishes to be integrated under CDEV. With dynamic binding and a name service resolution system at run-time, CDEV can dispatch an I/O request to the new service. Therefore, CDEV is not only a C++ toolkit which only offers predefined C++ classes but also a C++ framework within which new classes can be developed.

7 Concluding Remarks

CDEV is a C++ class library that provides a simple interface to control/DAQ services. It encapsulates an I/O request structure, event demultiplexing and synchronization mechanism of underlying services. It simplifies the development of control applications by treating all I/O events in the form of messages to devices and the task of integration of new services into CDEV by inheritance , dynamic binding and name resolution. Applications using CDEV will be easy to maintain and to port under any changes of services.

Currently, CDEV has a channel-access service layer, which has all the capabilities EPICS channel access API, tested extensively on HP- UX. There are several applications using CDEV currently in use at CEBAF including one new service, called the model server, which supports DIMAD [9] and other modeling engines. CDEV alone, without the channel access service layer, has been ported to SunOs, Ultrix, VMS and Aix operating systems. The source code for CDEV is available via anonymous ftp from ftp.cebaf.gov in pub/cdev. The web url is http://www.cebaf.gov/cdev.

8 Acknowledgment

CDEV development has been inspired by discussions within the EPICS community. Special thanks to Jim Kowalkowski, Claude Saunders and Marty Kraimer from the Advanced Photon Source at the Argonne National Laboratory for their valuable suggestions and thorough code reviews.

References

[1] Leo R. Dalesio, et. al., “The Experimental Physics and Industrial Control System Architecture: Past, Present, and Future”, *International Conference on Accelerator and Large Experimental Physics Control Systems*, Oct. 1993.

[2] William A. Watson III, Jie Chen, Graham Heyes, Edward Jastrzmbski, David R. Quarrie, “CODA: A Scalable, Distributed Data Acquisition System”, IEEE Trans.Nuc.Sci., Vol. 41, p61.

[3] Jeff Hill, “Channel Access: A Software Bus for the LAACS”, ICALEPCS, 1989.

[4] Bertrand Meyer. “*Object-Oriented Software Construction*”, Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[5] James Rumbaugh. “The life of an object model: How the object model changes during development”, *Journal of Object- Oriented Programming*, 7(1):24-32, March/April 1994

[6] Grady Booch, “*Object-Oriented Analysis and Design with Applications*”. Benjamin/Cummings, Redwood City, CA, 1994. Second Edition

[7] John Ousterhout, “*Tcl and the Tk Toolkit*”, Addison-Wesley, 1994.

[8] Douglas C. Shumidt and Tatsuya Suda. “The Service Configurator Framework: An Extensible architecture for dynamically configuring concurrent, multi-service network daemons”, In *Proceeding of Second International Workshop on Configurable Distributed Systems*, Pages 190-201, Pittsburgh, PA, March 1994. IEEE Computer Society.

[9] R. V. Servranckx, “*User's Guide to Program DIMAD*”, SLAC Report 285 UC-28 (A), MAY 1985.