BaBar Calibration Object Proposal

David Brown
12/8/96
draft 0.3

Introduction

In this document we propose a standard set of objects to process and store calibration information in BaBar. Implicit with these objects is a standard calibration programmer interface and a standard calibration interactive user interface. The purpose of this standardization is to allow the development of common tools for creation, verification, monitoring, and storage of calibration information. Common tools are desireable in order to minimize the redundant effort spent solving common problems in each subsystem. They also will allow the BaBar online to present a coherent calibration interface across the subsystems, minimizing the amount of expert knowledge needed to run and evaluate a full BaBar calibration.

A survey of current HEP experiments has shown that, while subsystems have similair calibration needs and end up with similair software, experiments rarely have common calibration software. Part of the reason for this appears to be that each subsystem wants to customize their data representation and their tools for their unique features. Another aspect is that common software arrives too late to be useful in prototyping and commissioning, so that subsystems develope their own software out of necessity. In response to this, a major goal of this proposal is to make it easy for subsystems to customize their calibration data and code within the framework. We also present this proposal early in the life of the experiment, so that subsystems can influence the design, and to provide them with a working tool while working with prototypes and commissioning.

The proposed design has been partially implemented with examples from the DIRC system. We have also implemented all the base classes, with a subset of the required common functions. A full list of the intended features (whether currently implemented or not) is given in appendix B.The implementation is purely in C++, and follows the basic rules of Object Oriented Design. The current implementation uses the Objectivity database, and the interactive program makes use of the BaBar offline framework. The code for the current implementation is publicly available from the BbrCalib package. Help from the subsystems is needed both to implement the subsystem-specific parts of the code, and to finish implementing the common functions.

Design Overview

The basic idea behind the proposed design is that calibration data consists of identical measurements made on a large number of channels. Each subsystem will have different numbers and groupings of channels, and will make different calibration measurements on these, but organization by channel is common. This implies the ability to use common collection classes to hold channel by channel information. Our proposal defines those common collection classes, plus the common functionality that does not depend on the specific nature of the channel data. These classes are based

on the RogueWave collection template classes, which are now standard to BaBar (and CERN) C++ development.

The implementation of the common collection classes relies on defining a C++ class for the calibration information of a single channel. Each subsystem will probably have several unique sub-classes (one for pedestals, one for gains, etc.), all inheriting from a common pure virtual base class. The data members of the specific sub-classes can be freely designed according to subsystem needs. The base class provides templates for all the required functions, provides the 'hooks' for the collection classes, and gives to each channel two data members, namely:

- An integer channel ID number
- A bitfield flag describing the channel condition

The channel ID can be defined as desired by the subsystem so long as it is unique for every channel within a collection object (the same ID can be reused within different collections). Some bitfields of the flag however will have a reserved common meaning, to describe universal concepts such as a 'dead','hot', or inactive channels. The remaining bits can be freely specified according to subsystem needs. The single channel sub-class objects would be restricted to having a fixed size, to allow their efficient storage in the Objectivity database[1] .

The proposed design recognizes three different but related environments relevant for calibration data, namely:

1. Online data-taking (Dataflow)
2. Data verification and monitoring (User interactive)
3. Database storage (Objectivity)

These environments represent different stages in the life cycle of calibration data, and imply different functional operations on that data. For example, calibration data is accumulated in the ROM in dataflow, examined in Unix in verification, and stored in the Objectivity database. To match these different functions, our proposal defines a different collection class for each environment, all representing the same data. The collection classes store a 'collection' of single-channel objects The base class for the channel objects allow (but do not require) use of the same sub-class in any of the three environments. For efficiency reasons, it may be desireable to concatonate channel information (IE store pedestals and gains together). This would be supported by the base class. Translation between the different collection representations is provided in a way that insures the consistency of the data contents accross the three environments in a user-transparent way.

Finally, our design recognizes that most operations one wants to perform on calibration data are common across subsystems, and so most can be implemented in the base classes. The common funcions include the ability to view the data, to histogram the data, to ntuple the data, to flag individual channels in the data, and to compare data with either an absolute or relative standard. Other functions include the arithemtic and logical manipulation of the data content such as averaging or merging of two sets. We also provide for the pedantic needs of reading/writing data from/to the database, and

[1] Variable sized objects in Objectivity must be stored as individual objects. The overhead of storing an individual Objectivity object for every channel of every subsystem is prohibitive.

accessing the data accumulated in the online DAQ. Subsystem customization of these functions is provided by either specific implementations of the single channel base class virtual functions, or through non-member functions which act on the single channel objects and are applied systematically to all members of the collection. The full description of anticipated functions is given in appendix A.

The following sections describe the design in some detail. A Booch diagram of the proposed classes is shown in figure 1. Appendix C describes the implementation of some of the proposed calibration objects specific to the DIRC. It should be emphasized that this design has not yet been fully implemented, and will undoubtedly develope as the code is written.

Single Channel Base Class

The base class for the single channel information is named CalChan. A typical daughter class inheriting from this might be called DircPedChan. The specific members of the daugter class might consist of the value of a calibration parameter (say the pedestal of the given channel) along with its RMS, and the number of samples used to compute these. The sub-classes must implement the pure virtual base class member functions, such as the 'print' function. A list of the single channel base class member functions is given in appendix B.

Collection Class Header

All the collection classes contain a header object of type CalHeader, which class is common across the three environments. The header provides the information which uniquely identifies the channel set it contains, in the form of a set name and number. For example, the header for the DIRC would have the name 'DIRC Sector', and the number of a particular sector (1 to 12). The divisioning of channels implied by the header should be roughly constant across the three calibration environments. In particular, random regrouping of channels in translating between two environments is not possible. The only supported regrouping will be concatonation, and that is supported only between the dataflow and the verification environments.

Each subsystem must decide on the proper grouping of their channels according to their specific needs. The guideline is to put together channels which are electronically 'near' to one another, for example all channels readout by a single ROM. Groupings of different types of calibration pertaining to the same set of channels (IE pedestals, gains, etc.) will be allowed by simple concatonation of the single channel objects. This type concatonation allows more efficient data storage and access.

The header also contains iformation about when the data were created and defines the current state of that data through a conditions flag. The data creation date may be inherited from a more general BaBar timestamp class. As with the channel flags, some of the set condition flag bits can be assigned by the subsystem, while others are common. The list of valid calibration data states is given in appendix C.

The header class can also be used as a base class, allowing the subsystems to customize storage of information common to a set of channels. For instance, a header

subclass might store the number of L1 accepts which were used to accumulate the data.

Dataflow Collection Class

As the BaBar dataflow user interface has not yet been defined, it is difficult to say exactly what the dataflow calibration collection class should look like. In its simplest form, this could be a fixed-length array of channel data objects, as for Objectivity storage (see below). Fixed length objects could allow a more efficient use of memory in the ROM. Some standard objects appropriate to this environment will be provided, allowing such common functions as keeping a running average and RMS, or filling a 'histogram'. These standard dataflow classes are defined in appendix C. Inheritance from these standard classes will allow subsystem specialization when the standard object functionality is not adequate.

Dataflow calibration objects would be created and transported by the BaBar dataflow system. Typically several objects of the same class will be used to cover the full channel count, as described above. Only a VxWorks user interface will be provided to these objects. At the UNIX level users will interface with the verification objects defined below, with the dataflow system providing a transparent translation between these.

Objectivity Collection Class

While Objectivity supports storage of variable-sized arrays, storage of fixxed size arrays is simpler. Since the number of channels in a system changes only rarely (if ever), it is thought to exploit this simplicity by restricting the calibration Objectivity collection classes to a fixxed size array of CalChan daughter objects. Calibrations which involve a unique CalChan daughter class will necessarily have a corresponding unique Objectivity collection class. It is possible that the Objectivity collection class for a particular subsystem/calibration could be identical to the corresponding dataflow class.

In our proposal the base class for Objectivity storage is called CalBank. Aside from defining their content, users will not interact directly with the CalBank objects. Instead, the calibration framework will provide a transparent translation of Objectivity objects to the corresponding verification objects described below. User access to the functionality of reading to and writing from the database will be provided as member functions of the verification class, as described below.

Verification Collection Class

Unlike the previous collection classes, the verification class is not a base class allowing further specialization through inheritance. Instead it is a general class capable of providing all the necessary functions for all subsystems and calibrations. Specialization of these functions is achieved through use of either CalChan virtual functions, or user supplied non-member functions which act on a specific CalChan daughter class. This mechanism allows the application of powerful tools in a general way, providing subsystem customization with a minimum of code duplication.

In our proposal the verification class is called CalList. It has a more extensive set of functions than the other two collection classes, as it provides the primary user interface,

and forms the bridge between dataflow and the database. To simplify the implementation of these member functions, heavy use is made of the STL container classes as implemented by RogueWave[2]. These organize the single channel objects into a linked list, and provide a number of standard access mechanisims (iterators, function application, sorting, etc.).

A special requirement for converting a verification object into its Objectivity form is that the set of channels be complete. Completeness must be defined by comparison with reference object. Intermediate manipulations of the verification object will not require completeness. Indeed, when composing sparse lists (such as a dead channel list), or lists of 'problem' channels to be printed out, completeness would be an unwanted feature. A full list of the verification class functions is provided in appendix A.

Conclusion

A proposal for a set of cooperating, common classes for implementing BaBar calibration in three environments (Datataking, verification, and storage) has been presented. A partial implementation of this proposal around the DIRC subsystem exists, and many of the common functions have also been implemented. Comments from subsystems and the BaBar computing management are welcome.

Appendix A

The base class used to store single channel data in the form of a RogueWave list, for verification and interacive access, is called a CalList. This will support the following functions:

1. Unary: All unary functions are member functions of CalList, and (in general) require a pointer to a function which operates on a CalChan.
    1. void histogram(double histvalue(CalChan*)); Histogram the result of a function returning a double applied to every CalChan element of the CalList.
    2. void ntuple(int N,void ntvalues(CalChan*,double*)); Fill an ntuple with one entry per channel, where the ntuple has the specified number of columns (not counting the ID number and flag, which are always present). The 'ntvaules' function fills an array of doubles from the channel contents, which is then used to fill the ntuple.
    3. void print(ostream&); print the contents of the bank. This recursively calls the virtual CalChan 'print' function, creating a tabular output either to the screen or a file.
    4. void statistics(ostream&); Calculate statistics for the current bank. This evaluates and then displays the mean and RMS for all the data elements.
    5. void settag(tagval); This sets the tag in the header.
    6. void setchantag(tagval); This sets the tag on all the contained CalChan elements
    7. void addchantag(tagval); This 'or's the existing CalChan tag with the new value
    8. bool save(); This translates the CalList to its ODBBank representation, and saves that in the database according to the input parameters. The CalList is tested for completeness before being written, using the 'verify' function below.
    9. void cut(bool select(CalChan*,void*),void*); This reduces the channel content to only those CalChan objects for which the user-supplied 'select' function returns 'true'.
    10. double average(bool select(CalChan*,void*),double value(CalChan*),void*); This returns a single value representing the 'average' of the data for all CalChan elements for which the user-supplied 'select' function returns 'true'. The averaging is performed as a 'running average' on the return of the 'value' function applied to the selected elements. This function is used in the strip-chart capacity described below.
    11. CalList(const CalList& copy); This creates a copy of a list, appending a 'copy from' record to the history.
2. Binary: All binary functions are CalList member functions, and take as input a pointer to another CalList object, and possibly a function which operates on two CalChan objects. Functions which alter the set of channel data return a new CalList object, those which alter the data elements make copies instead of changing the existing elements.
    1. CalList* compare(CalList,bool compare(CalChan*,CalChan*,void*),void*); This returns a new CalList

[2] The "Tools.h++" library of classes by RogueWave is the BaBar standard implementation of the "Standard C++ Library" '

whose CalChan data element consists of a single boolean value, set according to the return value of the user supplied ‘compare’ function.

2. CalList* select(CalList*); This removes those channel elements whose IDnumber is not present in the second CalList.
3. CalList* antiselect(CalList*); This removes those channel elements whose IDnumber is present in the second CalList.
4. CalList* merge(CalList*,bool which(CalChan*,CalChan*,void*),void*); This returns a new CalList whose contents are a merger of the contents of the two supplied CalLists. If a CalChan object for a particular channel ID is present in only one of the two input CalLists, it is automatically copied to the output CalList. If both CalLists contain a CalChan object for a particular channel ID, the one copied is selected as a result of the ‘which’ function applied to each CalChan object, ‘true’ meaning copy the first CalList’s element, ‘false’ meaning copy the second CalList’s element, where first and second are defined according to the order of the argument list. Supplying the trivial function ‘true’ always selects the CalChan object from the first CalList whenever both contain a CalChan object with the same ID. Invoking the ‘merge’ function on two CalLists containing independent channel lists using the ‘true’ function serves to concatonate the channel contents of two lists into a single list.
5. CalList* average(CalList*,CalChan* average(CalChan*,CalChan*)); This resets the channel data elements to the ‘average’ of the data elements of the two banks. The user-supplied ‘average’ function returns a pointer to a new CalChan whose data in some way averages the two inputs. The output bank will only contain elements for which both CalLists had a CalChan object of the same ID number. The two CalLists must be of the same type (IE have compatible headers).
6. CalList* concatonate(CalList*,CalChan* average(CalChan*,CalChan*)); This resets the channel data elements to the ‘concatonation’ of the data elements of the two banks. the user-supplied ‘concatonate’ function returns a pointer to a new CalChan whose data in some way concatonates the two inputs. Unlike the ‘average’ function, the two CalChan objects do not have to be of the same type.
7. bool complete(CalList*); Compare the header and the channel IDs of all the list members with the input reference set, and return ‘true’ if these are equivalent. This function uses only the CalChan and CalHeader base class functions, and does not compare daughter CalChan data members.

3. interface functions
    1. stripchart; Make a stripchart of the time behavior of a group of channels. Inputs specify the time range, the database specification of the objects, and a function which returns a double for each occurance of the database object.

Appendix B

The proposed base class for channel data is called a CalChan. Its data members consist of an integer identification (channel) number, as well as a 32-bit flag. Functions operating on these members are provided in the base class, while functions operating on subsystem specific members are defined only as virtual functions. The following member functions must be written for the daughter CalChan classes.

1. virtual void print(ostream&); Output stream operator for printing the contents.
2. virtual CalChan* copy(); Make a copy of the specific daughter class, and return a (CalChan*) pointer to it.

The following functions are not members, but can be friends of the CalChan daughter classes. They are used in calls to the CalList functions

1. bool select(CalChan*,void*); Unary selection function. The void* can point to any data structure, which can be used as desired in the evaluation.
2. bool compare(CalChan*,CalChan*,void*); Binary comparison function, retruning ‘true’ if the objects are sufficiently similair to be considered equivalent, false otherwise. The void* structure can be used to specify the parameters o f the comparison.
3. int bool which(CalChan*,CalChan*,void); Binary comparison function, returning ‘true’ to select the first object, false to select the other.
4. double histvalue(CalChan*); The output value is suitable for histograming.
5. void ntvalues(CalChan*,double*); The output array of doubles is suitable for ntupling.

The proposed class for collection class headers is called CalHeader. Its data members consist of a date and time stamp (which is probably supplied by a BaBar standard header), a name and number defining the data set it describes, and a condition flag.

Appendix C

Describe the generic classes and the dirc-specific classes here.