BaBar Calibration Object Proposal

David Brown
1/26/97
draft 0.9

Introduction

In this document we propose a standard set of objects to create, process and store calibration information in BaBar. Implicit with these objects is a standard programmer interface (calibration toolkit). The purpose of using standard objects is to minimize the redundant effort spent solving common problems in each subsystem. Standardization will also provide a coherent calibration interface across the subsystems, minimizing the expert knowledge needed to run and evaluate a full BaBar calibration. While this proposal deals explicitly only with online calibration, it is hoped that the same objects and interface will be useful for offline extraction of calibration constants from the physics data. That possibility will be discussed in a later note.

Many ideas presented in this document are a synthesis of information obtained through an informal survey of calibration software and practices at several current HEP experiments (D0, CDF, and ALEPH). Much of the verification functionallity is copied directly from the DBANA program from CDF. One observation from examining the calibration systems at other experiments is that they have achieved varying degrees of standardization, apparently due to historical reasons. A major goal of this proposal then is to provide a standard framework early in the life of the experiment, so that the subsystems can provide design input, and can start using the software in prototype tests and commissioning.

The proposal described in this document has been partially implemented, using the DIRC as an example subsystem. A full list of the intended features is given in appendices A and B. The implementation is purely in C++, and follows the basic rules of Object Oriented Design. The current implementation is based on the Objectivity database, though that is hidden from the user interface. The code for the current implementation is publicly available through the SLAC cvs system as the BbrCalib (general code) and DrcCalib (DIRC specific code) packages.

This note is concerned with the processes which culminate in the storage of calibration information into the conditions database. The related issues of the dataflow and reconstruction interfaces for reading back calibration information is not treated in this document.

As with all online processes, conditions information for calibration datataking and verification (run setup parameters, verification cut values, etc.) must be persistently stored. This storage mechanism and interface will not be unique to calibration, and so is not discussed in this document.

Finally, since this proposal has not yet been fully implemented or accepted by either the BaBar subsystems or online management, the design presented should be considered as provisional, and will undoubtedly change (as will this document).

Design Overview

The proposed design is based on a few main ideas:

1. Calibration data mainly consists of identical measurements made on a large number of channels.
2. Different types of measurements are needed on the same channels (pedestals, gains, etc.), but the data formats are simple, and many of the same functions are needed on all calibration types.
3. A subsystem consists of many channels, and it generally makes more sense to deal with collections of channels rather than individual channels.
4. There are three life stages to calibration data:accumulation, verification, and storage. These are associated with three environments (accumulation=dataflow, verification=UNIX, storage=conditions database/Objectivity), which imply different restrictions on coding practice and data formats. The calibration framework should provide a coherent interface in all three environments.
5. User code should not directly access the conditions database, both to minimize the possibility of persistent data corruption, and to insulate users from possible storage technology changes.
6. Subsystems must provide two types of calibration user code: dataflow feature extraction, and verification analysis. The former will be governed mainly by the general dataflow rules, while the later is defined by the calibration framework.
7. Some calibration information is global (not particular to a single channel). A means for efficiently and securely storing associated global information with individual channel information must be provided.
8. Sometimes it is necessary to know how calibration constants were derrived in order to correctly interpret their value. This history must be kept in machine and human readable form, for both individual channels and channel collections, as part of the persistant store of the data.

Points 1 and 2 imply that a single, simple base class can be used to describe calibration data from a single channel for all types of calibration for all subsystems. By keeping this class simple, it can be made to operate in all three environments listed in point 4. This single channel calibration class forms the backbone of this proposal, and it is discussed in detail later.

Points 2 and 3 imply the need for an organizational class which describes the channel collection objects for different calibration types. This organizational class will also define the verification and other data conditions. To be compatible with dataflow, channel collections will be constrained to contain all channels from one or more ROM (IE, ROM channels can't be sub-divided or re-partitioned). Further details of the collection objects are provided below.

Point 4 implies that the channel collections will be implemented differently in the different environments. The calibration framework will internally hide these differences to present a uniform interface to the user.

Points 5 and 6 imply the need for two calibration user interfaces: one with dataflow, the other with the verification environment. The Objectivity interface will be hidden inside of verification. Verification might occur automatically in the OEP part of dataflow, or it may be performed interactively on a dedicated workstation: in both cases, the programmer interface will be the same, and assume a UNIX environment. Both the verification and dataflow interfaces are described in more detail below.

Points 7 and 8 imply respectively the need for a calibration collection header class and history class. These classes are described below.

Single Channel Calibration Base Class

A pure virtual base class (CalChan) is provided as the parent from which all classes describing calibration data for a single channel must inherit. This class serves several purposes:

1. It is the root class used in making collections of channels.
2. it defines the (pure virtual) functions which allow the channel collections to implement a uniform interface.
3. It defines two data members, namely:
    - A channel logical ID number
    - A channel condtion flag

The channel logical ID is a 32 bit integer which uniquely identifies every channel in BaBar, including those used in testbeams and test stands. It is based on the 16 bit dataflow logical ID[1] , which describes the position in Datafow down to the ROM level. Subsystems are free to define the remaining 16 bits as they please, though this definition must be unique for a given 'experiment' value. Subsystems may provide global functions which translate from full channel logical ID to physical ID, and vice versa, when the subsystem hardware numbering scheme is not based on the dataflow system (the SVT readout section is an example). Other functions which depend on channel logical ID are described in the section below on collection objects.

The condition flag is meant to describe the channel status for a particular type of calibration (this is discussed again in the collection object section below). Some fields of the flag will have a reserved common meaning, to describe universal concepts such as a dead,hot, or inactive channels. The remainder of the flag is free to be defined as desired by the subsystems, perhaps differently for different calibration types. CalChan provides a virtual function which translates the common fields into text strings, which can be overwritten to also translate the user fields. The condition flag can describe multiple conditions at the same time, through allocation of bit fields within it. The condition flag is currently implemented as a 32 bit integer, with 16 bits reserved and 16 bits user-defined. The currently defined common field flag values and text strings are described in appendix A.

The CalChan class contains no members for storing actual calibration data; those will belong to the daughter class. Access to the calibration data is however provided at the base class level through virtual functions. These functions provide the number of data elements, and give each a name, unit, and value range, and provide access to their value. Through these functions, a CalChan object is self-describing in a fashion similair to an EPICS channel. A list of the CalChan virtual functions is provided in Appendix A.

[1] See 'The Dataflow System',BaBar note # xxx, Mike Huffer etal

Single Channel Calibration Derrived Classes

The calibration data members of a CalChan daughter class may be composed of any fixed size combination of primitive data types (either long or double). The fixed size requirement is necessary to allow efficient storage of collection objects in the Objectivity database[2] . No pointers are allowed for the same reason, but fixed length arrays are possible, as are structures which themselves are of fixed size and composed of primitive data types. The CalChan data access functions must however be implemented to return primitive data types, even when the direct object owned by the CalChan daughter is an array or structure.

CalChan daughter classes should be named according to the convention XXXT...TChan, where XXX is the subsystem three letter acronym, and T...T describes the calibration type (see the section on calibration types below). Some CalChan daughter classes can skip the XXX field, if they are used in more than one subsystem (a generic pedestal class PedChan might be an example). A typical daughter class might be called DrcT0Chan (DIRC subsystem, T0 calibration type), and have data members consisting of the T0 value and its error. Some example CalChan daughter classes for the DIRC are given in appendix C.

Each subsystem is required to define a CalChan daughter class for describing the global status of every channel in their subsystem, with name XXXStatusChan. This object will have no data members outside those provided by the base class, but will implement all the functions (many with trivial implementations). The common fields of the condition flag will define the status of this channel to all users of the subsystem (such as dataflow). The values of this flag should combine the status flags from the calibration types of that subsystem in whatever way they deem fit. The XXXStatusChan class is used in the status objects described below in the collection class section.

Calibration Type Class

In this proposal, individual channel calibration objects are not free-standing, and must be part of achannel collection object to be used. During verification, collections of any number and type of channels are allowed (see the Verification Interface section below), but only pre-defined objects may be stored in the conditions database, in order to maintain its integrity and useability. To facilitate the pre-definition of valid calibration collection objects, a special class CalType, which is itself persistant, is used. CalType is a base class whose daughters describes the conditions a channel collection object must meet before it can be stored. For any given period, exactly one CalType object should be present in the conditions database for each type of calibration a subsystem wishes to store.

The CalType class has to perform several functions:

1. Define a unique name for this calibration type, which is its own (subclass) name. The name should be composed as XXXT...TType, where XXX is the three letter subsystem acronym, and T...T is the 'type' name (Ped,Gain, etc.). DrcT0Type is a valid example.
2. Identify the particular CalChan subclass required for collection objects of this

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

type. Note that there need not be a 1↔1relation between CalChan and CalType subclasses.

3. Define the number of objects of this type necessary to fully cover all channels of the subsystem. If 'N' objects are needed, those objects will be numbered from 1 to 'N'.
4. Unambiguously assign every channel logical ID to an object (IE a number between 1 and 'N'). This is performed by a member function.
5. Define the number of CalChan objects necessary to complete a given object (this can be different for different objects of the same type).
6. Define the conditions which each individual channel must satisfy before the collection object containing it can be stored. This is performed by a function which checks the CalChan status flag, whose value must have been previously set.

As mentioned before, collection objects are not allowed to contain a partial set of channels from a given ROM; they must either all or none be contained. Collection classes containing channels from more than 1 ROM are allowed, however their use will be restricted to the offline. A type which is intended to be used online (for instance in feature extraction) must define collection objects which have a 1↔1 correspondence with the ROMs, to allow them to be downloaded.

It is likely that some calibration measurements will be repeated using different techniques; for example, constants first extracted from a dedicated online calibration run might be refined later using the physics data stream. In this case, it is natural to use the same CalChan object to define the channel data in both cases. The collection objects built from the two calibrations are then distinguishable only by being associated to different CalType objects. The CalType objects would be identical except for two fields:The type name, and the precedence flag. The different precedence flag will allow users to select one or the other collection object according to criteria which they define (in some cases the online calibration object may be more appropriate, in others the offline). Arbitration between collection objects representing different measurements of the same quantity is performed by the CalType 'alias' function.

Calibration Header Class

A base class CalHeader is provided to store information which is global to a collection object. All collection objects own a CalHeader object. CalHeader is a free-standing class containing basic information, but it can also be subclassed to include global information particular to a given calibration type. In this case, the subclass name should be XXXT...THeader, where XXX is the subsystem three letter acronym, and T..T is the calibration type name (Ped, Gain, etc.).

Calibration History Class

The history of a collection class is stored in the form of a CalHistory object, which is owned by every collection class. History information is stored in the form of a list of integer codes. Every operation performed on a collection object which alters its data contents (creation, verification, etc.) must also make an appropriate entry in the history list. A (persistant) dictionary translating the integer codes into text strings will be provided, with an entry for all the base class functions already provided. In addition to the code, each data altering operation will record the time when the operation was performed and, if a second collection class is involved, the time stamp of that second object.

Dataflow Interface

Because the dataflow system does not yet exist, even in prototype form, we are unable to provide an implementation. Instead, we present here our ideas as to how the calibration will interface with dataflow. It should be noted that calibration is a special case of a dataflow user, and must obey all the protocalls described in the dataflow manual.

The dataflow interface provided by calibration is implicit in the collection class for that environment. The channel collection object in the dataflow environment will consist of a simple array of CalChan objects. Calibration types which are intended (either now or eventually) to be implemented in the ROM must define their objects with a $1\leftrightarrow1$ correspondence with the ROM. During a calibration data run, objects of a valid type can be created in a particular ROM simply by specifying the type: the dataflow system will provide the information as to which ROM is concerned, and thereby the specific object.

A number of the single channel calibration function appropriate to the dataflow environment are common to all subsystems: for instance, averaging a value over a range of L1-accepts, or histogramming that value. To facilitate use of common code for these common functions, a small number of worked-out CalChan daughter classes appropriate to dataflow will be provided. These are listed in appendix A.

Verification Interface

The verification interface is provided by the member functions of a single class, the CalList class. Unlike the previous collection classes, CalList is not a base class but instead is a general class. Specialization of the CalList 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.

CalList 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 RogueWave tools.h++ template classes. The current implementation of a CalList is as a single linked list (RWTPtrSlist). A complete list of the CalList member functions is given in appendix B.

Conditions Database Interface

As mentioned above, the conditions database interface is hidden from the user through the verification interface. This latter provides both read and write access. Read access is intended to facilitate comparisons between collection objects whose data were taken at different times. The write access is restricted in that only complete and verified collection objects can be written to the database. The definition of 'verified' is left to the subsystems. The interface will allow any user to write valid and complete collection objects to the database. However, the ability to introduce new calibration types to the database, or to change the validity rules of an existing type, will require the use of a password. This password will be given to the calibration specialist of each subsystem, and will permit that person to introduce types pertinent only this his/her subsystem. The subsystem calibration specialist may retain/distribute the password as they like.

In spite of being hidden, the reader may wish to know that the current (Objectivity) implementation of the conditions database is built around an ooVarray of CalChan objects. This provides efficient storage for a variable number of simple objects. Many of the coding restrictions above derrive from having to ultimately translate a channel collection object into an ooVarray.

Conclusion

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

Appendix A

The proposed base class for single channel calibration data is called a CalChan. Its data members consist of a 32 bit integer identification (channel) number, as well as a 32-bit integer status flag. Functions operating on these members are provided in the base class, while those pertaining to daughter class calibration data content are defined only as virtual functions. The following are CalChan member functions:

1. virtual void print(ostream&)= 0; //Output stream operator for printing the contents.
2. virtual int numValues() = 0; // number of calibration data values
3. virtual char* valueName(int) = 0; //Name for the specified value
4. virtual char* valueUnit(int) = 0; //Unit of the specified value
5. virtual long* valueLong(int) = 0; //Pointer to integer value of the value, =0 if not an integer
6. virtual double* valueDouble(int)= 0; // pointer to floating value of the value, =0 if not a floating point
7. virtual long* valueLongLowLimit(int) = 0;// Lower integer limit to the value, =0 if not an integer
8. virtual long* valueLongHighLimit(int) = 0;// Lower integer limit to the value, =0 if not an integer
9. virtual double* valueDoubleLowLimit(int) = 0;// Lower integer limit to the value, =0 if not an integer
10. virtual double* valueDoubleHighLimit(int) = 0;// Lower integer limit to the value, =0 if not an integer
11. virtual CalChan* copyOf() = 0; //Make a copy of the specific daughter class, and return a (CalChan*) pointer to it.
12. bool verify(void*) = 0; // verify the channel calibration data member contents

The following are function prototypes (IE the name is not fixed) which are used in calls to the CalList functions. Specific versions of these may be made 'friend' to specific CalChan daughter classes.

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. 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. This
5. void ntvalues(CalChan*,double*); The output array of doubles is suitable for ntupling.

The CalChan base class includes a status flag member. 16 bits of this are reserved, as described below:

1. bits 0-1: channel status. 0= OK, 1=Not OK, 2=BAD

2. bit 2: dead channel bit. Dead channels return the same output to any input stimulus.
3. bit 3: hot channel bit. Hot channels return random,large signals a large fraction of the time.
4. bit 4: inactive channel bit. No data from this channel will (should) appear in the data stream.
5. bit 5:suppress bit. Signals from this channel should be ignored.
6. bit 6:unoriginal data. The data stored in this CalChan object do not come from the same source as that described in the collection header.
7. bit 7:unverified bit. This bit defines whether the CalChan 'verify' function has been run on the channel, regardless of the result of that verification test. This is the only bit set automatically in all new CalChan objects, and can only be unset by running the 'verify' function.

In addition to the base class, the calibration framework also provides a few fully worked out generic CalChan objects. These can be used as part of any collection object, transient or persistent, though some are more appropriate for certain environments than others. Below is a list of the worked-out classes, along with their specific members.

- MeanChan. This class keeps a running mean and RMS of a single quantity, and is intended mainly for use in dataflow. It has the following members:
  - double value; // current value
  - double rms; // current rms
  - double nsamples; // current number of samples
  - void increment(double value); // increment the running values
  - void reset(); reset the running values
- Hist50Channel. This class keeps a 50-bin histogram of a single quantity. The histogram limits are defined in a HistHeader object (daughter class to CalHeader).
  - double histo[50]; // bin contents
  - void fill(double value); // increment the histogram
  - void reset(double value); //reset the histogram contents

Appendix B

The verification environment channel collection class is called CalList. This is a single class, and is not subclassed for specific use. Instead, it uses the virtual interface to generic CalChan objects to implement its functions in a subsystem independent way. CalList stores its CalChan objecst in a RogueWave list. It supports the following functions, divided into catagories as unary(operations on a single list) or binary (another list is required as input).

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. int histogram(double histvalue(CalChan*)); Histogram the result of a function returning a double applied to every CalChan element of the CalList. the return value is the histogram ID number
   2. int histogram(int); Histogram the nth calibration value of the CalChan. If the value is intrinsically an integer, it is first converted.
   3. 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.
   4. void ntuple(); Fill an ntuple with one entry per channel, where a columns is assigned to each calibration values in the CalChan object.
   5. void print(ostream&); print the contents of the list. This recursively calls the virtual CalChan 'print' function, creating a tabular output either to the screen or a file.
   6. void statistics(ostream&); Calculate statistics for the current bank. This evaluates and then displays the mean and RMS for all the data elements.
   7. void settag(tagval); This sets the tag in the header.
   8. void setchantag(tagval); This sets the tag on all the contained CalChan elements
   9. void addchantag(tagval); This 'or's the existing CalChan tag with the new value
   10. bool save(CalType*,int); This translates the CalList to its ODBBank representation, and saves that in the database as object 'J' of the type specified by the specified CalType object. The CalList is tested for completeness and channel validity using the rules of the supplied CalType object before it is stored.
   11. 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'.
   12. 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.
   13. CalList(const CalList& copy); This creates a copy of a list, appending a 'copy of ...' 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 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 C

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