# Calibration Constants Database (CCDB) package documentation

GlueX-doc-2625-v2

Mark M. Ito, Dmitry A. Romanov
Thomas Jefferson National Accelerator Facility
12000 Jefferson Avenue
Newport News, VA 23606

June 15, 2012

**Abstract**

This note contains a documentation of the CCDB package. A package for storing and managing calibration constants database.

## Contents

**1 Introduction** **3**

**2 Basic concepts** **4**
2.1 Data structure . . . 4
2.2 Namepath . . . 4
2.3 Variation . . . 4
2.4 Variations hierarchy . . . 5
2.5 Requests . . . 6
2.6 When full data request should NOT be used . . . 7
2.7 Default values . . . 8
2.8 Connection strings . . . 9
2.9 Update and delete operations . . . 10

**3 CCDB command line tutorial** **11**
3.1 Getting started . . . 11
3.2 Help system . . . 12
3.3 Commands . . . 12
3.3.1 Commands consistency . . . 12
3.3.2 Commands overview . . . 12

**4 C++ API** **14**
4.1 C++ Overview . . . 14
4.2 C++ User API . . . 14

**5 Python API** **15**
5.1 Python Overview . . . 15
5.2 Using CCDB from python . . . 15

**6 Command Line Interface Reference** **16**
6.1 Options . . . 16

**7 Appendix A** **16**

# 1 Introduction

Calibration Constants Database (CCDB) aims the next goals:

- Storing calibration constants.
- Managing calibrations.
- API for JANA, plain C++, Python.
- Additional Logging, import, export data.

CCDB stores data as tables with columns and rows. As a data storage CCDB supports:

- **Naming**. Each table is identified by path-name;
- **Versioning**. Each table may has many versions of data;
- **Branching**. So called "variations" allows to use branches of data;

As a management tool and as a data provider CCDB allows:

- C++ User API. Allows an easy access to CCDB data from C++.
- JANA API. An integration to JANA framework.
- Python API. Allows accessing and managing CCDB from python language.
- Command line tools. Tools to manage CCDB data from the shell.
- Web interface.

# 2 Basic concepts

CCDB basic usage concepts and CCDB console tool for managing CCDB contents

## 2.1 Data structure

## 2.2 Namepath

The desired set of values is identified by name. The name string is unique across all detector systems. The convention already implemented in the GlueX code base is to use a forward slash(/) notation to specify a hierarchical namepath.

For example:

`/FDC/driftvelocity/timewalk_parameters`

This allows implementors of individual detector systems to to specify a hierarchy with as much or little depth as is needed given their complexity. The "/FDC/driftvelocity/timewalk_parameters" parameters may have members identified by either name or position. For example, it may contain 3 values: "slope", "offset", and "exponent". By contrast, a set of constants with a namepath "/FDC/CathodeStrips/pedestals" may have 100 values identified simply as "0", "1", "2", "3", ...

**Namepath format**:
Allowed symbols are a-z, A-Z, 0-9, _ and -. There is no space or special symbols allowed in the namepath. Such considerations simplifies the console management and database validation of namepath objects.

```
/My-path/to/data_01            #Ok

/Some...thing/is wrong here!   #ERROR illegal symbols
```

## 2.3 Variation

The CCDB includes a *variation* feature. The variation is a sort of data branching, making a "shallow copy" of a complete set of calibration constants (i.e. one that refers to constants without actually copying them).

The variation is to be used primarily to create a new set of constants based on another set, but with a few changes. A variation is identified as a string. The primary purpose of a variation is to allow testing of new constants or alternative sets of constants that don't disturb the main trunk and don't require copying a complete set.

Specifying a is optional for end-users. If no variation is specified, then a "*default*" variation is used. It is assumed that the default variation will be used to hold the best available constants at any point in time.

Each variation is stored in the CCDB with information regarding another variation upon which it is based. This basis variation may specify the default variation.

Each set of constants in the CCDB includes a variation string that is used to specify which variation those specific constants are valid for.

**Variations format**:
Allowed symbols are a-z, A-Z, 0-9, _ and -. There is no space or special symbols allowed in variation name. Such considerations simplifies the console management and database validation of variation names.

## 2.4 Variations hierarchy

Variations have hierarchy. The idea behind variation hierarhy is simple: if one wants to experiment with calibration data without changing it one creates a variation(thus branching the data). If one wants to go deeper and experiment with changed data without changing it more, one create a child variation.

The top level is "default" variation. When any new variation is created a parent variation may be specified. If user doesn't specify the parent, new variation is created with "default" as a parent. Thus one can say, that "default" variation is "super" or "root" parent.

Lets examine how variation hierarcy work by real example.

1. Lets say we have 3 data tables with names: "table1", "table2", "table3". Each table has some default data: "data1", "data2", "data3" in it.

   | Table \ Variation | default |
   |---|---|
   | table1 | data1 |
   | table2 | data2 |
   | table3 | data3 |

   This means that when somebody call

   ```
   calibration->GetData("table1",...)
   ```

   - "data1" is returned.

2. Now lets bring variations to the scene. Mr John wants to experiment with constants in table2. So he creates a variation named "john". When a variation is created it should have some parent. If no parent is explicitly specified, the "default" variation is set as parent.

   So now we have a variations hierarchy:

   ```
   "default" <- "john"
   ```

   Mr John adds data to "table2" in the "john" variation. Thus now we have:

   | Table \ Variation | default | john |
   |---|---|---|
   | table1 | data1 | |
   | table2 | data2 | data2.1 |
   | table3 | data3 | |

   For all regular users, who doesn't specify that want to work with John's data, a call:

   ```
   calibration->GetData("table2"...) //returns "data2".
   ```

   But If one runs JANA with environment variable

   ```
   JANA_CONTEXT="variation:john"
   ```

   Then data from john variation will be used whenever it is possible. So when "table2" is requested, "data2.1" is returned. But when table1 is requested, there is no specific data in john variation for table1, so CCDB goes looking data in parent variation which is the "default" variation. So data1 is returned.

   So once again: if one runs JANA with `JANA_CONTEXT="variation:john"`, then requests:

   ```
   calibration->GetData("table1"...) //returns "data1"
   calibration->GetData("table2"...) //returns "data2.1"
   calibration->GetData("table3"...) //returns "data3"
   ```

3. Bill wanted to use John's "data2.1" but change table3 data. Bill creates "bill" variation with "john" variation as parent. So now we have a hierarchy:

```
"default" <- "john" <- "bill"
```

Bill adds "data3.1" set of constants to "table3" and variation "bill".

| Table \ Variation | default | john | bill |
|---|---|---|---|
| table1 | data1 | | |
| table2 | data2 | data2.1 | |
| table3 | data3 | | data3.1 |

Lets see how requests will work if one specifies "bill" variation. If "table3" is requested, then "data3.1" is returned. If "table2" is requested, there is no data for table3 and "bill" variation, so CCDB goes looking for "bill" parent, which is "john" variation. "john" variation has "data2.1" for "table2" so "data2.1" is returned. If "table1" is requested, there is no other data than in "default" variation which is on top of variation hierarchy, so "data1" is to be returned. Summarizing:

```
JANA\_CONTEXT="variation:bill"
calibration->GetData("table1"...) //returns "data1"
calibration->GetData("table2"...) //returns "data2.1"
calibration->GetData("table3"...) //returns "data3.1"
```

## 2.5 Requests

There are two problems CCDB tries to solve:

1. Generally, getting constants should be as easy as to say "Give me this constants for June 2011"
2. There must be a way to uniquely identify every set of data. [1]

CCDB uses so called "Requests" to solve both of the problems.

Full form of the request is

```
</path/to/data>:<run>:<variation>:<time>
```

Full form of the request [2] gives uniquely identify a set of data. In database terms the full request is an "unique composite key" for the particular data values.

But parts of a request could be omitted. The minimal request to get the data is just /path/to/data One may omit any part of the request except for name-path.

Lets look at examples:

- /path/to/data - no run, variation or timestamp is specified
- /path/to/data::mc - no run specified, variation is "mc", no date is specified
- /path/to/data:::2029 - only the path and the date(year) are specified

As shown in the examples above, to specify a path and a variation but to use default run one skips the run number and leave its place like "::"

```
                +-- variation
                |
                |
```

[1] "integer indexes" is the first thing that comes to mind when one hears words "unique key" and "database". Why don't CCDB use it? But CCDB is database independent. Indexes, which are good to use with relational databases, become uneasy with standalone ASCII files or ROOT files. Moving indexes from one database to another might be a problem. And, last but not least, indexes are good for machines but not for humans. If human operator has an index 1114211 it tells nothing to him and it could be easily mistaken with 1142111 which belongs to absolutely another data set.

[2] with full form of time section (given up to seconds)

```
/path/to/data::mc
              ^
              |
              +-- place where run number should be
```

And the request '/path/to/data:::2029' means that a path and a date are specified but a run number and a variation are left to be set by default. What does 'default' means? It is discussed further in "DEFAULT VALUES" chapter.

The time is parsed as: **YYYY:MM:DD-hh:mm:ss**

Any non digit character may be used as separator instead of ':' and '-'

so all next lines are parsed the same

```
2029/06/17-22:03:05
2029-06-17-22-03-05
2029/06/17:22/03/05
2029a06b17c22d03e05
```

One can omit any part of the time string starting from the right, this the latest date for this part will be returned.

Examples:

"2011" - (this means the year 2011), it will be interpreted as 2011/12/31-23:59:59 timestamp so the latest constants for year 2011 will be returned.

"2012/05/21" - it will be interpreted as 2012/05/21-23:59:59 meaning to be the latest constants for 21 May 2012

CCDB searches the closest constants before or equal to timestamp provided.

## 2.6 When full data request should NOT be used

The short answer to this question is:
In application code. If you have a full request in your application code **in production environment** it is definitely NOT how CCDB is designed.

The long answer:
Usually C++ code that reads out data from CCDB looks like this:

```
calibration->GetCalib("/CDC/general_params");
```

The run number is provided by application, and user may specify variation and date through command line or environment variables. Everything is counted by CCDB when 'calibration' object is created. By following design user may change variation or time without recompiling the application.

But CCDB C++ (and python) API accepts full version of requests for debugging and sketch up purposes. Next code works and run number, variation and time given by application is overwritten by 100:my_variation:2011 for this transaction.

```
calibration->GetCalib("/CDC/general_params:100:my_variation:2011"); //FOR DEBUG ONLY
```

Please, remove everything beyond first ":" when you clean up the code and get it ready for production use.

If you use full requests in some places of production version of the application, users may not know about it and be surprised when they change variation using application flags but data is changed everywhere except the spots with full requests.

## 2.7 Default values

There are two general cases of using the requests:

1. To read out data in physics software
2. To manage constants (using CCDB CLI, python, etc.)

In the first case, physics software usually knows and provides a run number being processed. Also the software is responsible for allowing users to set a default variation or(and) constants date for the analysis. (The way the software interacts with CCDB is described in C++ API and Python API chapters)

CCDB Command Line Interface (CCDB CLI) is the main facility to manage constants. The default values could be set by flags and commands that are described in CCDB command line tutorial section. But generally it sets run=0, variation='default' if no flags are given.

So, CCDB defaults and priorities (1 - highest)

**Run number:**

1. Run number specified in a request
   If one uses "/path/to/data:100" request, constants for run 100 are returned regardless of the run being processed
2. Software set run number.
   If 10200 run is being processed and one requests "/path/to/data", then data for run # 10200 is returned.
   If one starts CCDB CLI as "ccdb -i -r 10200", the run is used as default for the session.
3. 0 - (means run number 0).

**Variation:**

1. Variation specified in a request
   If one uses "/path/to/data::mc" request, the variation 'mc' is used to return the data. [3]
2. Global preferred variation set by software
   *If one starts CCDB CLI as "ccdb -i -v mc", variation 'mc' is used as default for the session*
3. The "default" variation

**Timestamp:**

1. Request specified time will be used
2. Current time

When one uses CCDB console tool in interactive mode, one can set the default run number by running 'run' command

Example:

```
> run 100
> cat /path/to/data     # all commands will get constants for run 100
> run                   # you can check what run is set by default
100
```

[3] Using variation in request doesn't cancel variation hierarchy feature. Lets say request is "/path/to/data::mc" and variation 'default' is parent of 'mc' variation ('default' → 'mc'). If there is no data that has been specially added for 'mc' variation for table '/path/to/data', then CCDB looks for constants of parent variation, i.e. 'default' variation. See Variations hierarchy chapter.

C++ API section will overview how to set default run

-v start flag and var command to set working variation -r and 'run' command sets default run for ccdb session So...

```
ccdb -i -v mc -r 100 #runs ccdb shell with variation 'mc' and run 100 by default

>run       # findout ccdb default run number
100

>var       # findout default variation
mc

>run 200   # change run default run nubmer during session
>run
200

>var john  # change variation during session
>var
john
```

## 2.8 Connection strings

CCDB uses so called "connection strings" to specify a data source. The generic format of a connection string is:

```
<protocol>://<datasource specified string>
```

SQLite connection string:

```
sqlite://<path to sqlite db file>
```

MySQL connection string:

```
mysql://<username>:<password>@<server_address>:<port>/<database>
```

One may omit any part except "mysql://" and "¡server_address¿". The default values will be used.

CCDB MySQL connection defaults:

- username - ccdb_user
- password - no password
- port - default MySQL port (now is 3306)
- database - ccdb

Here is the order of how ccdb gets the connection string:

1. The default connection string is **"mysql://ccdb_user@localhost ccdb"**
2. if **CCDB_CONNECTION** environment variable is set it is used overwriting the default connection string
3. if -c or –connection flag is given in command prompt it is used overwriting all other.

Example 3. Connection string 1:

```
"mysql://john@localhost:999"
```

- MySQL server on 'localhost' using port 999
- user is 'john' with no password

- the database is 'ccdb' by default

Example 4. Simple connection string:

```
"mysql://localhost"
```

- MySQL server on localhost using port 3306 (default)
- user is 'ccdb_user' with no password (default)
- the database is 'ccdb' (default)

Example 5. Full connection string:

```
"mysql://smith:hHjD83f@192.168.1.3:4444/ccdb_database"
```

It tells ccdbcmd to connect to:

- MySQL server on '192.168.1.3' using port 4444
- user is 'smith' with password 'hHjD83f'
- the database is 'ccdb_database'

## 2.9 Update and delete operations

CCDB follows two principles in terms of updating and deleting:

1. Don't delete anything.
2. Updates are done by '*adding new*'.

If one wants to *update* the values of some table, it is done by *adding a new* set of constants. The software will use more recent set of data by default.

If one wants to change the number of columns of the table (or columns specification), it is done by adding a new table with a new name.

For example. If one wants to change the format of a table:

```
/FDC/driftvelocity/timewalk_parameters
```

One should create a new table with right format:

```
/FDC/driftvelocity/timewalk_parameters2
```

CCDB doesn't provide any tools for deleting ***non empty*** tables or directories, changing their names or specifications at user level.

There is a strong reason for this limitation. If CCDB is used to hold calibration constants, then some code should exist for each data set. At least somewhere at some point of time. So deleting or changing something (the number of rows in a table for example) will lead to corruption of the code. The worst case scenario is when such changes don't lead to immediate crush but produce weird hard-to-determine bugs somewhere in deep parts of an offline software.

(Deleting an empty table is OK - there is no data so no code behind it)

If one really need to delete some constants, this should be done at *administrative level*. This means that is should be discussed, users should be notified about changes, and the changes should be tested. Thus deleting the constants shouldn't become a normal every day experience for users.

# 3 CCDB command line tutorial

This section is a tutorial of using CCDB command line tools.

## 3.1 Getting started

CCDB provides command line tool for introspection and management of constants database. To access it call 'ccdb' shell command (CCDB should be installed and its environment variables are set)

'ccdb' can be used as an interactive shell or as a single command.

Usage from command line:

```
ccdb <ccdb arguents> command <command arguments>
```

Usage as interactive shell:

```
ccdb <ccdb arguments> -i
> command1
> command2
> ...
> q
```

Example 1. Command line mode:

```
           (1)                          (2)    (3)
ccdb -c "mysql://john@localhost:999" ls /TOF/params
```

1. **-c "mysql://john@localhost"** - sets the ccdb connection string. If -c flag is not given, ccdb will try CCDB_CONNECTION environment variable, if CCDB_CONNECTION default connection string. The connection strings are described in 2.8
2. ls - is a ccdb command which returns a list of directories and tables that belongs to directory '/TOF/-params'
3. /TOF/params - is the argument of ls command. Like a posix shell ls.

Example 2. Interactive mode:

```
ccdb -i -c "mysql://john@localhost:999"                (1)
> ls /TOF/params                                       (2)
> help                                                 (3)
> cd /TOF                                              (4)
> cd params
> ls
> pwd                                                  (5)
> q                                                    (6)
```

1. flag '-i' will start ccdb in interactive mode.
2. 'ls /TOF/params' - the result of the is exactly the same as in Example 1. One stays in the interactive shell after the execution.
3. 'help' command provides list of commands and how to use each of them
4. executing next commands will reproduce Example 1 step by step.

5. The same as in posix shell, ccdb interactive mode have the current working directory, with relative and absolute paths. pwd command shows the current working directory.
6. to exit interactive mode enter 'q', 'quit' or press ctrl+D

Since ccdb objects have /name/paths and many other things that looks like POSIX file system, the commands are very posix-shell-like.

## 3.2 Help system

The ccdb is designed to be a self descriptive. By using 'help' 'usage' and 'example' commands one could get all the commands and how to use them.

By using 'howto' command one could get tutorials for typical situations.

## 3.3 Commands

### 3.3.1 Commands consistency

Command keys are consistent. This means that some flags and argument formats are the same across all commands. There are unified flags to identify objects for all commands:

- **-a** - Assignment
- **-v** - Variation
- **-t** - Data table
- **-r** - Run or run-range
- **-d** - Directory

For example '**info**' command may be executed against directory, table or variation. Example 6. Info command:

```
[bash promt] ccdb -i
     > info -v default                                    (1)
     > info -r all                                        (2)
     > info -d /TOF                                       (3)
     > info -t /TOF/params                                (4)
     > info /TOF/params                                   (5)
```

1. Get information about "default" variation 2. Get information about "all" runrange. "all" runrange is [0, infinite_run] 3. Get information about "/TOF" directory. 4. Get information about "/TOF/params" type table 5. By default '*info*' treat non flag argument as a name of a table.

### 3.3.2 Commands overview

This table is printed if one executes "ccdb help"

Assuming that user is in interactive mode, one may categorize the commands:

*To navigate directories* pwd - prints current directory cd - switch to specified directory ls - list objects in the directory (wildcards are allowed) mkdir - creates directory

Example 7. Directory commands overview:

| | | |
|---|---|---|
| info | Info | Prints extended information about an object |
| vers | Versions | Show versions of data for the specified table |
| run | CurrentRun | Gets or sets current working run |
| dump | Dump | Dumps data table to a file |
| show | Show | Shows type table data |
| mkdir | MakeDirectory | Create directory |
| pwd | PrintWorkDir | Prints working directory |
| cd | ChangeDir | Change current directory |
| add | AddData | Add data constants |
| mktbl | MakeTable | Create constants type table |
| cat | Cat | Show assignment data by ID |
| ls | List | List objects in a given directory |

Table 1: List of ccdb commands

```
    > pwd
    /
> cd /TOF
> ls
    table1  table2
> mkdir constants
> ls con*
    constants
```

**Get information about objects**

- **info** - gets information about objects (use -v -r -d flags), see example 6.
- **vers** - gets all versions of the table
- **cat** - displays values
- **dump** - same as cat but dumps files to disk
- **logs** - see logs information

**Manage objects**

- **mkdir** - creates directory
- **mktbl** - creates data table
- **add** - adds data from text file to table (variation and runranges are created automatically by add command)

# 4 C++ API

## 4.1 C++ Overview

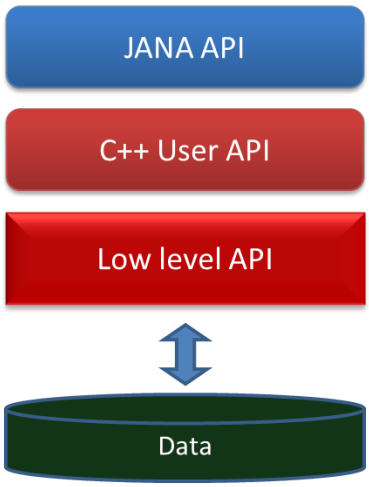


Figure 1: C++ API levels

CCDB C++ library consists of separate levels (see fig. 1).

- **C++ User API** - This level is *most probably what any user*, who is not bound to JANA, *needs*. The API provides simple functionality of getting constants. It automates connections number, multi-threading, data source selection, etc.
- **Low level API** - provides all sorts of low level functionality for managing data and CCDB internals. This level considered to be internal and could be interesting only in terms further development of CCDB. Regular users should not use any classes from this level [4].
- **JANA API** - There is a JANA API on top of Users C++ API. This level integrates CCDB and JANA framework.

## 4.2 C++ User API

There are two main classes in C++ User API:

- **CalibrationGenerator**
- **Calibration**

The **Calibration** is used to get constants. The **CalibrationGenerator** is used to get **Calibration** instance.

[4]Low level API is subject to change for better CCDB performance and stability. The changes are done without any remorse towards user's code which is happened to use Low level API elements. That is the reason why Low level API should be kept separated and used only in CCDB higher level classes

# 5 Python API

## 5.1 Python Overview

- **CalibrationGenerator**
- **Calibration**

## 5.2 Using CCDB from python

One can use CCDB directly from python using ConsoleContext class. PYTHONPATH is automatically adjusted when one source environment.bash CCDB script and is set to

```
PYTHONPATH="$CCDB_HOME/python":"$PYTHONPATH"
```

How to use it:

```
import ccdb.path_utils
import ccdb.cmd.themes
from ccdb import get_ccdb_home_path
from ccdb.cmd.console_context import ConsoleContext

#Initialization
#============================

#connection strings:
ccdb_path = get_ccdb_home_path()
sqlite_connection_str = "sqlite:///" + os.path.join(ccdb_path, "sql", "ccdb.sqlite")
mysql_connection_str = "mysql://ccdb_user@127.0.0.1:3306/ccdb"

#create console context, this is the main class
context = ConsoleContext()

#set all exception to be raised and propagated instead of just going to log
#so you can try-except them
context.silent_exceptions = False

# disable colored output
context.theme = ccdb.cmd.themes.NoColorTheme()

# set connection string
context.connection_string = sqlite_connection_str

# your username for logging
context.user_name = "python_tests"

# Register all commands (ls, rm, mktbl etc...)
# One can use context after calling this function
context.register_utilities()

#now you can use context to run ccdb command
try:
context.process_command_line("mkdir /test/testable2 x y z #Some comment")
```

```
...

Please look at
$CCDB_HOME/python/tests/test_console_context.py
There are a lot of examples of how to use console_context class

Dmitry
```

The reasons for that comes from intention

# 6 Command Line Interface Reference

## 6.1 Options

**-p** Prompt for MySQL server password. The server may be configured to require one.

# References

# 7 Appendix A

Closed connections automatically reopened if needed. CalibrationGenerator have UpdateInactivity function which goes through all opened calibrations and disconnects ones that is idle for long. Time of idling is controlled through SetMaxInactiveTime, GetMaxInactiveTime functions.

```
//Initialization part:
CalibrationGenerator calibGen;
calibGen.SetMaxInactiveTime(100);  // 100 seconds

//...

//Each event or on timer:
calibGen->UpdateInactivity()
```

Possible future outline:

1. introduction
2. design logic
3. ccdb shell interface
4. C++ API
5. jana API
6. examples
   (a) user
   (b) calibrator
7. future work

There is a script at $CCDB_HOME/scripts/mysql2sqlite/mysql2sqlite It takes all flags from mysqldump and creates a sqlite file.

Sample command:

```
mysqldump -hhalldweb1 -uccdb_user ccdb | sqlite3 ccdb.sqlite
```