A Proposal for Super-Table Generation Software

**Authors**

For the Fermilab LAFS Team: Elliott McCrory and Terri Lahey

For BE/OP: Mario Pereira

Version 2

18 June 2010

A framework for calculating the individual cells of the LHC super-table is defined here. This is a draft document.

# framework Features

The Framework consists of these three processes:

1. *Synchronous to the LHC Cycle*: Get time-relevant data from the LHC.
2. *Twice for an LHC Fill (at the beginning of Stable Beams and at the end):* Re-arrange and perform calculations on the gathered data. Output to a super-table storage (database?). There are three outputs per cell of the Super-Table, and they are all strings:
	1. Output number 1: The value of the cell. Cannot be “null”.
	2. Output number 2: The units of this value (which can be “null”)
	3. Output number 3: An explanation of how this calculation was done.  Cannot be “null”, and should be rather long and detailed!
	4. Output number 4: An error message if the calculation failed (hopefully, this is always “null”)
3. *At the conclusion of Process #2*: Generate static HTML and Excel-ready files.

By “time-relevant,” we mean that a set of data is collected from a device at a specified transition into a Beam Mode.

For clarity, here is a cartoon of the architecture.

## Definitions

What is a framework? If you read the Wikipedia article on “Software Frameworks”[[1]](#endnote-1) we are not making one. Process #2 is almost a “framework”. We’ll use that term anyway.

# Process #1: Gather Correct Data from the LHC

This section is to be completed by Mario.

# Process #2: Creating the Super-Table Database

This process breaks into two sub-processes:

1. Generating the actual super-table cell data
2. Saving these data into the Super-Table database

The architecture of step “b” is described in [another document](Data%20Storage%20Schema%20for%20the%20LHC%20SuperTable.docx). The elements of the archiving are:

* Interface STSave: For saving the archive information
* Interface STColumn: For specifying the columns in the Super-Table.
* Interface STFills: For obtaining the valid fill numbers that go into the Super-Table

## Architecture for Generating the Super-Table Cell Data

We define this interface:

* Interface STCell: The classes that are responsible for calculating the contents of a specific cell in the super-table will implement this interface. It is initialized with the fill number and the column description. It returns the value, the Units and the algorithm.

There is also a class that will cause the data for a fill to be calculated (or re-calculated). This class is called STStart. Also, there is a class that provides a graphical user interface to STStart, which we will call STStartApp. There also will need to be other classes for initializing the Super-Table database.

### Public Interface STCell

The basic calculation of each cell is implemented in an object that implements this interface:

**public** **interface** STCell {

 **public** **void** begin(**int** fill, **String** column);

 **public** String getValue();

 **public** String getUnits();

 **public** String getAlgorithm();

 **public** String getErrorMessage();

 **public** **void** end(**int** fill, **String** column);

}

It is guaranteed that each of these six methods will be called in this order for each instance of STCell, unless the “begin” method throws an exception.

As a trivial example, one would implement column 1, the Fill Number, in this way:

**public** **class** FillNumber **implements** STCell {

 **private** **int** fill;

 @Override

 **public** **void** begin(**int** f, **String** column) {

 **this**.fill = f;

 }

 @Override

 **public** **void** end(**int** f, **String** column) {}

 @Override

 **public** String getAlgorithm() {

 **return** "The fill number is defined elsewhere and simply returned";

 }

 @Override

 **public** String getUnits() {

 **return** **null**;

 }

 @Override

 **public** String getValue() {

 **return** "" + fill;

 }

 @Override

 **public** String getErrorMessage() {

 **return** **null**;

 }

}

The column header (parameter “column”) is included in the “begin” and “end” methods so that a single class can be used for several columns. For example, the fill begin, fill end and fill duration columns are implemented in a single class. The transverse emittance columns could be a single class, too.

Note: In this architectural model, the class is instantiated separately for each cell. Static attributes would be used to hold common information. It would not be hard to change this so that a single instance of each class is instantiated.

### Public Class STStart

The class that directs the calculation of each cell is presented here:

**public** **class** STStart {

 **private** Class<? **extends** STCell>[] classes = **new** Class[]{

 FillNumber.**class**, /\* 1, Fill Number \*/

 FillDuration.**class**, /\* 2. Begin Fill \*/

 FillDuration.**class**, /\* 3. End Fill \*/

 FillDuration.**class**, /\* 4. Fill Duration \*/

 /\* … \*/

 };

 **private** **int** fill;

 **private** STSave archiver;

 **public** STStart(**int** fill) {

 **this**.fill = fill;

 initialize();

 run();

 save();

 }

 **private** **void** initialize() {

 archiver = STFactory.*getSave*();

 }

 **private** **void** run() {

 **int** column = 0;

 **for** (Class<? **extends** STCell> C : classes) {

 **try** {

 STCell cell = C.getConstructor().newInstance();

 cell.begin(fill, columnHeaders[column]);

 archiver.setValue(column, cell.getValue());

 archiver.setUnits(column, cell.getUnits());

 archiver.setAlgorithm(column, cell.getAlgorithm());

 archiver.setErrorMessage(column, cell.getErrorMessage());

 cell.end(fill, columnHeaders[column]);

 } **catch** (Exception e) {

 String errorMessage = Calculation of column " + column +

 " failed due to an exception: " + e.getMessage());

 System.err.println(errorMessage);

 e.printStackTrace();

 archiver.setErrorMessage(column, errorMessage);

 }

 column++;

 }

 }

 **private** **void** save() {

 **try** {

 archiver.write(fill);

 } **catch** (IOException e) {

 e.printStackTrace();

 }

 }

}

(The actual implementation contains more checks. They are removed here for clarity.)

Notice that an array of classes, which each implement STCell, is traversed to build the super-table.

An instance of each class is created in the standard way:

 STCell cell = C.getConstructor().newInstance();

If it was required that a single instance of each STCell class be instantiated, the array classes would become an array (called, say, “stCellObjects”) and the “C.getConstructor.newInstance()” would be called in the creation of this array, rather than during the cell calculation.

Notice that every exception is caught, noted and then ignored. This should insure that every cell has a value, even if it is “N/A”. [Note: This code needs to be extended to insure this criterion fully.]

While the list of classes is contained in a hard-coded array here, it could easily be moved to a file or a database table.

# Process #3: Creating the Static HTML Files

This process creates the appropriate HTML and Tab-Delimited files to satisfy the requirements for displaying the Super-Table. It creates the following 24 files:

1. All Fills, HTML.
2. All Fills, TAB.
3. The “Best 10” fills, HTML. Currently, we use the Tevatron criterion for “best”: the largest initial luminosity, averaged over the two main experiments.
4. The “Best 10” fills, TAB.
5. The most recent 10 fills, HTML.
6. The most recent 10 fills, TAB.

These six files are created for these four “Levels” of display:

1. **Level I**: Global information and summary information at the declaration of “Stable Beams”. The data are averaged over all bunches, both accelerators, or averaged over all experiments
2. **Level II**: Global information and summary information at the declaration of “Stable Beams”. These data are the same as the Level I data, except that the luminosities from all the experiments are included separately.
3. **Level III**: Global information and summary information at each Beam Mode state transition. The data are averaged over all bunches, over both beams, or over all of the experiments.
4. **Level IV**: Global information and detailed information at each Beam Mode state transition. These data are averaged over all bunches, but the accelerators are show separately.

The architecture must be flexible enough to add more views without much difficulty. For example, it may be that a fifth level would be added that includes the RMS variation in the beam parameters that are averaged over all the bunches.

# References

1. Taken from the Wikipedia on 26 May 2010, <http://en.wikipedia.org/wiki/Software_framework>. [↑](#endnote-ref-1)