**RBSP ECT**

**Science Operations Center**

**DBProcessing: database setup for new missions, products, and codes**

**Date: April 25th, 2012**

**Brian Larsen**

**Los Alamos National Laboratory**

**Revision - A (Draft)**

**Changes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Description of Change** | **Approved by** | **Date** |
| A | Preliminary Draft Version |  | 04/24/2012 |

**Table of Contents**

1.0 Purpose 5

2.0 Required information 5

2.1 Mission 5

2.1.1 Mission name 5

2.1.2 Mission root directory 5

2.2 Satellite 5

2.2.1 Satellite Name 5

2.3 Instrument 5

2.3.1 Instrument name 5

2.4 Product 5

2.4.1 Product name 6

2.4.2 Relative path 6

2.4.3 Format 6

2.4.4 Level 6

2.5 Process 6

2.5.1 Process name 6

2.5.2 Output product 7

2.6 Code 7

2.6.1 Filename 7

2.6.2 Relative path 7

2.6.3 Start date 7

2.6.4 Stop date 7

2.6.5 Description 7

2.6.6 Process 7

2.6.7 Interface version 7

2.6.8 Quality version 7

2.6.9 Revision version 7

2.6.10 Active code 8

2.6.11 Date written 8

2.6.12 Output interface version 8

2.6.13 Arguments 8

2.7 Inspector 8

2.7.1 Filename 9

2.7.2 Relative path 9

2.7.3 Description 9

2.7.4 Interface version 9

2.7.5 Quality version 9

2.7.6 Revision version 10

2.7.7 Output interface version 10

2.7.8 Active code 10

2.7.9 Date written 10

2.7.10 Arguments 10

2.7.11 Product 10

3.0 Setup worksheet 11

# Purpose

Provide instructions on adding a mission to the RBSP ECT SOC processing chain. This is a living document that will be updated as the process changes. Currently the process is quite manual and advances will be made to make it easier in the future.

While this document is specific to RBSP ECT processing, the processing chain is designed to be general and in principle can be used to implement the processing chain of ANY satellite project that consists of defined processing steps, data products and processing codes.

# Required information

## Mission

This is the highest level in the database; a mission is something like GPS or RBSP.

### Mission name

The mission must have a unique name. 20 character max

### Mission root directory

The mission must have a predefined root directory, create this directory. This does have to be unique. 50 character max

## Satellite

A satellite is the physical entity collecting the data, for example NS41 in GPS or RBSP-A in RBSP.

### Satellite Name

Every satellite for a given mission must have a unique name. 20 character max

## Instrument

An instrument sits on a satellite and collects the data, for example HOPE on RBSP-A in RBSP.

### Instrument name

Every instrument for a given satellite must have a unique name. There is no requirement that satellites in a mission have the same instruments. 20 character max

## Product

A product is a set of identical formatted files containing data of a specified format. A product can be directly created by an instrument or derived as output from a process involving one or more input products.

### Product name

A product name must be unique for each instrument. The file names for RBSP follow an agreed naming convention (RBSP-filename-document-R2.docx). In general, filenames are of the form:

<source>\_<type>\_<descriptor>\_<date>\_<version>.cdf

The product name corresponds to the <descriptor> in the filename. Examples are filename

rbspa\_pre\_ect-hope-L1\_0272\_v1.0.0.cdf

which has the product name “ect-hope-L1”.

Product names are 30 character max.

### Relative path

The path within the mission root directory where this product will be stored. This does not need to be unique [untested], but maybe should be. 50 character max

### Format

A format string used to build output file names from processes. Currently recognized formats are defined in the file DBStrings.py. The current formats recognized are:

* Y: 4 digit year
* m: 2 digit month
* b: 3 character month (Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
* d: 2 digit day
* y: 2 digit year
* j: 3 digit day of year
* H: 2 digit hour (24-hour time)
* M: 2 digit minute
* S: 2 digit second
* MILLI: 3 digit millisecond
* MICRO: 3 digit microsecond
* QACODE: the QA code (ok|ignore|problem)
* VERSION: version string, interface.quality.revision
* DATE: the UTC date from a file, same as Ymd
* MISSION: the mission name from the db
* SPACECRAFT: the spacecraft name from the db
* PRODUCT: the product name from the db

No max size.

### Level

The level of the file, this is a float for generality. Examples 0, 1, 2, 1.5

## Process

A process is a code that takes N inputs products and outputs 1 new product. Processes are assigned to an output product.

### Process name

The name of the process, this must be unique for a mission. 20 character max

### Output product

The product that the process will output. Can only be one.

## Code

A code is the actual executable code that is responsible for a process. All codes have the limitation of taking in N arguments and output 1 file whose name is specified on the command line. Additionally arguments can be specified in the db that are passed for each call. There is no limitation on language or implementation so long as N files in and 1 specified file out as arguments.   
If a code does NOT conform to this format then a wrapper must be written for the code that does take N input files and 1 output file as arguments, and calls the underlying routines correctly.

### Filename

The filename of the code. This is only the base filename without any path information. 50 character max

### Relative path

The relative path within the mission root directory where the code resides. 50 character max

### Start date

The first date for which the code is valid. Files with earlier dates will not be processed by this code.

### Stop date

The last date for which the code is valid. Files with later dates will not be processed by this code.

### Description

A brief description of what the code does, who wrote it, or other useful information. 50 character max

### Process

The process that the code belongs to. There can only be one code per process.

### Interface version

The interface version of the code.

### Quality version

The quality version of the code.

### Revision version

The revision version of the code.

### Active code

Boolean argument if the code is used or not

### Date written

The date the code is written, adds information for the admins to debug possible issues.

### Output interface version

The interface version of the file written by this code.

### Arguments

Additional arguments to the code. This is useful if the same software creates multiple output data products by use of optional arguments. For example this can be switch for the software to output science data or engineering data. No max size.

## Inspector

An inspector is user supplied code that identifies files that are a certain product and fills in various information that only the user knows and the database requires.

An inspector is a piece of Python 2.6-2.7 code that identifies if a file is a particular product and fills in required information needed by the db.

An inspector must meet these requirements:

1. An inspector **shall** import inspector
2. An inspector **shall** import Version as is must instantiate an Version class
3. An inspector **may** import Blogging and add comments to the processing log files as desired. This is recommended.
4. An inspector **shall** be named Inspector and subclass inspector.inspector

*class Inspector(inspector.inspector):*

1. An inspector **shall** define the class variable code\_name containing a descriptive string such as a the filename.
2. An inspector **shall** define a method named inspect that when called fills in the required variables below. The use of any other methods, functions, or classes is allowed.  
   *def inspect(self, kwargs):*
3. An inspector inspect method **shall** return None is the file does not match the inspector. This includes any malformed files.
4. An inspector **shall** set the variable self.diskfile.params['utc\_file\_date'] with a Python datetime.datetime object
5. An inspector **shall** set the variable self.diskfile.params['utc\_start\_time'] to the earliest time covered by the file (Python datetime.datetime object)
6. An inspector **shall** set the variable self.diskfile.params['utc\_stop\_time'] to the latest time covered by the file (Python datetime.datetime object)
7. An inspector **shall** set the variable self.diskfile.params['data\_level'] to a float value representing the data level of the file. This must match what is specified in product.
8. An inspector **shall** set the variable self.diskfile.params['version'] to a instantiated Version object.
9. An inspector **may** set the variable self.diskfile.params['verbose\_provenance'] to any string information to be associated with the file.
10. An inspector **may** set the variable self.diskfile.params['quality\_comment’] to any short comment about the file’s quality.
11. An inspector **may** set the variable self.diskfile.params['caveats'] to any short string containing caveats about the file.
12. An inspector **may** set the variable self.diskfile.params['release\_number'] to any integer related to a release number of the data file.
13. An inspector **may** set the variable self.diskfile.params['met\_start\_time'] to any integer related to the first MET value covered by the file. It is up to the user to keep this and self.diskfile.params['utc\_start\_time'] consistent. self.diskfile.params['met\_start\_time'] is not used in calculations.
14. An inspector **may** set the variable self.diskfile.params['met\_stop\_time'] to any integer related to the last MET value covered by the file. It is up to the user to keep this and self.diskfile.params['utc\_stop\_time'] consistent. self.diskfile.params['met\_stop\_time'] is not used in calculations.
15. An inspector inspect method **shall** return anything other than None upon filling in the required variables. A string is a recommend return value.

### Filename

The filename of the inspector, this is the base name with no path information. 50 character max

### Relative path

The relative path under the mission root directory the inspector resides. 50 character max

### Description

Any useful information about the inspector. 50 character max

### Interface version

The interface version of the code.

### Quality version

The quality version of the code.

### Revision version

The revision version of the code.

### Output interface version

Not currently used.

### Active code

Boolean if this inspector is currently in use.

### Date written

The date the inspector was written, adds information for the admins to debug possible issues.

### Arguments

Additional arguments to the inspector. This is useful if the same inspector code works for multiple products by use of optional arguments. For example if filename can differentiate then a regular expression might be passed in. No max size

### Product

The product that the inspector is associated with, this must be a one-to-one mapping.

# Setup worksheet

Fill out this worksheet as progressing through the process to aid in the addition of a new chain to the db.

Mission name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Mission root directory: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Manually setup the root directory
2. Enter mission name and Mission root directory into database
   1. import DBUtils2
   2. dbu = DBUtils2.DBUtils2(‘Test’)
   3. dbu.\_openDB()
   4. dbu.\_createTableObjects()
   5. dbu.addMission(<mission name>, <root directory>)
   6. dbu.\_closeDB()
3. Note the mission ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
4. Instantiate a DBUtils2 object for the new mission
   1. import DBUtils2
   2. dbu = DBUtils2.DBUtils2(<mission name>)
   3. dbu.\_openDB()
   4. dbu.\_createTableObjects()

Satellite name: ­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. add the satellite to the db
   1. dbu.addSatellite(<satellite name>)
2. Note the satellite ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Instrument name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Add the instrument to the db
   1. dbu.addInstrument(<instrument name>,<satellite ID>)
2. Note the instrument ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Product name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Product relative path: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Product filename format: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Product level: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Add the product to the db
   1. dbu.addProduct(<product name>, <instrument id>, < Product relative path>, None, <Product filename format>, <Product level>)
2. Note the Product ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
3. Add the instrument product link (ties products to instruments, can be more than one)
   1. Dbu.addInstrumentproductlink(<instrument id>, <product\_id>)

Inspector filename: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inspector relative path: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inspector description: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inspector version: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inspector is active: Yes/No

Inspector date written: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inspector arguments: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Inspector product: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Add the inspector
   1. dbu.addInspector(<inspector filename>, <inspector path>, <inspector description>, <version>, <inspector active True/False>, <date written>, <inspector output interface version>, True, <product id>)
2. Note the inspector ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Process name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Output product: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Add an output product to the db (see product)
2. Add the process to the db
   1. dbu.addProcess(<process name>, <output product ID>, None)
3. Note the process ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
4. Add product process link (sets up inputs to processes)
   1. dbu.addproductprocesslink(<input product id>, <process id>)

Code filename: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code relative path: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code start date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code stop date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code description: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code process: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code version: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code is active: Yes/No

Code date written: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code output interface version: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Code arguments: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Add the code to the db
   1. dbu.addCode(<code filename>, <code relative path>, <code start date>, <code strop date>, <code description>, <process id>, <version>, <code active True/False>, <date written>, <code output interface version>, True, <code arguments>)
2. Note the code ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_